

Aztec C UniTools for the Amiga

**Version 5.0
October 1989**

Copyright 1988, 1989 by Manx Software Systems, Inc.
All Rights Reserved
Worldwide

DISTRIBUTED BY:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

USE RESTRICTIONS

You are permitted to install and use this product on a single computer. Multiple CPU systems require supplementary licenses.

Before using any Aztec C68k products, the License Registration included with this product must be signed and mailed to:

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702

COPYRIGHT

This software package and document are copyrighted ©1988, 1989 by Manx Software Systems. All rights reserved worldwide.

No part of this publication may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without prior written permission of Manx Software Systems.

DISCLAIMER

Manx Software Systems makes no representations or warranties with respect to this product and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Manx Software Systems reserves the right to modify the programs and revise the contents of the manual without obligation to notify any person of such revision or changes.

TRADEMARKS

Aztec C68k, Manx AS, Manx LN, Z, and SDB are trademarks of Manx Software Systems. CP/M-86 and CP/M-80 are trademarks of Digital Research. MSDOS is a trademark of Microsoft. PC DOS is a trademark of IBM. UNIX is a trademark of AT&T Bell Laboratories. Macintosh and Apple II are trademarks of Apple Computer. Atari is a trademark of Atari Computers. Amiga is a trademark of Commodore-Amiga.

Manual Revision History

October 1989	Second Edition
--------------	----------------

December 1988	First Edition
---------------	---------------

Table of Contents

Chapter 1 - Overview

Introduction	1 - 1
About This Manual	1 - 2
Technical Support	1 - 3

Chapter 2 - Diff

The Conversion List	2 - 1
Conversion Items	2 - 2
The Command Line	2 - 2
The Affected Lines	2 - 3
The -b Option	2 - 4
Differences Between the UNIX and Manx Versions of diff	2 - 4

Chapter 3 - Grep

Description of grep	3 - 1
OPTIONS	3 - 3
INPUT FILES	3 - 3
PATTERNS	3 - 4
Differences Between the Manx and UNIX Versions of grep .	3 - 7
OPTION DIFFERENCES	3 - 7
PATTERN DIFFERENCES	3 - 7
Examples	3 - 7
SIMPLE STRING MATCHING	3 - 7
MORE EXAMPLES	3 - 8

Chapter 4 - Make

Preview	4 - 2
Note to Previous Users	4 - 2
The Basics	4 - 2
What make Does	4 - 3
The Makefile	4 - 3
RULES	4 - 6
make's Use of Rules	4 - 6
Example	4 - 7
Interaction of Rules and Dependency Entries	4 - 8
Advanced Features	4 - 8
Dependent Files	4 - 8
MACROS	4 - 9
Using Macros	4 - 10
Defining Macros in a makefile	4 - 10
Defining Macros in a Command Line	4 - 11
Macros Used by Built-In Rules	4 - 11
Special macros	4 - 11
RULES	4 - 12
Rule Definition	4 - 12
Built-in Rules	4 - 14
COMMANDS	4 - 14
Allowed Commands	4 - 14
Logging Commands and Aborting make	4 - 15
Long Command Lines	4 - 15
Makefile Syntax	4 - 15
Comments	4 - 16
Line Continuation	4 - 16
Starting make	4 - 17
The Command Line	4 - 18
make's Standard Output	4 - 18

Differences between the Manx and UNIX make Programs . . .	4 - 19
Examples	4 - 20
The makefile in the 'libc' Directory	4 - 21
Makefile for the 'sys' Directory	4 - 22
Makefile for the 'misc' Directory	4 - 23

Chapter 5 - Z Editor

Requirements	5 - 2
Components	5 - 3
Getting Started	5 - 3
CREATING A NEW PROGRAM	5 - 3
The Screen	5 - 4
Modes of Z	5 - 4
Insert Mode	5 - 4
Exiting Z	5 - 5
EDITING AN EXISTING FILE	5 - 5
Starting and Stopping Z	5 - 6
Command Line Options	5 - 6
The Cursor	5 - 7
Moving Around in the Text: Scrolling	5 - 7
Moving Around in the Text: the Go Command	5 - 8
Moving Around in the Text: String Searching	5 - 8
Finely Tuned Moves	5 - 9
Deleting Text	5 - 9
More Insert Commands	5 - 10
Summary	5 - 10
More Commands	5 - 11
THE SCREEN	5 - 11
Displaying Unprintable Characters	5 - 11
Displaying Lines That Do Not Fit on the Screen	5 - 11
Commands	5 - 12

Special Keys	5 - 12
PAGING AND SCROLLING	5 - 12
SEARCHING FOR STRINGS	5 - 13
Additional String-Searching Commands	5 - 13
Regular Expressions	5 - 14
Enabling Extended Pattern Matching	5 - 15
LOCAL MOVES	5 - 16
Moving Around on the Screen:	5 - 16
Moving within a Line	5 - 16
Word Movements	5 - 17
Moves within C Programs	5 - 18
Marking and Returning	5 - 19
Adjusting the Screen	5 - 20
MAKING CHANGES	5 - 20
Small Changes	5 - 20
Operators for Deleting and Changing Text	5 - 21
Deleting and Changing Lines	5 - 22
Moving Blocks of Text	5 - 23
Duplicating Blocks of Text: the Yank Operator	5 - 23
Named Buffers	5 - 24
Moving Text between Files	5 - 25
Shifting Text	5 - 26
Undoing and Redoing Changes	5 - 26
INSERTING TEXT	5 - 26
Additional Insert Commands	5 - 27
Insert Mode Commands	5 - 27
Autoindent	5 - 28
MACROS	5 - 28
Immediate Macro Definition	5 - 28
Examples	5 - 29
Indirect Macro Definition	5 - 31
Reexecuting Macros	5 - 31

Wrapping Around During Macro Execution	5 - 32
EX-LIKE COMMANDS	5 - 32
Addresses in Ex Commands	5 - 33
The Substitute Command	5 - 34
The c Option	5 - 34
The g Option	5 - 34
Examples	5 - 35
The "&" (Repeat Last Substitute) Command	5 - 35
QUICKFIX COMMANDS	5 - 36
THE OPTION FILE	5 - 36
The ZOPT Environment Variable	5 - 37
Setting Options for a File	5 - 37
STARTING AND STOPPING Z	5 - 37
Starting Z	5 - 38
Starting Z without a Filename	5 - 38
Starting Z with a List of Files	5 - 38
Stopping Z	5 - 39
ACCESSING FILES	5 - 39
Filenames	5 - 40
Writing Files	5 - 40
Reading Files	5 - 41
Editing Another File	5 - 41
File Lists	5 - 43
Tags	5 - 44
The ctags Utility	5 - 45
OPTIONS	5 - 45
DIFFERENCES BETWEEN Z AND VI	5 - 47
Command Summary	5 - 49
Starting Z	5 - 49
The Display	5 - 49
Options	5 - 49

Adjusting the Screen	5 - 49
Positioning within File	5 - 50
Marking and Returning	5 - 50
Line Positioning	5 - 50
Character Positioning	5 - 51
Words and Paragraphs	5 - 51
Insert and Replace	5 - 51
Corrections During Insert	5 - 52
Operators	5 - 52
Miscellaneous Operations	5 - 52
Yank and Put	5 - 52
Undo and Redo	5 - 53
Macros	5 - 53
QuikFix Commands	5 - 53
Colon Commands	5 - 53

OVERVIEW

DIFF

GREP

MAKE

Z EDITOR

OVERVIEW

1

Chapter 1 - Overview

Introduction

Welcome to UniTools--the perfect supplement to your Aztec C software development system! The utilities in UniTools work with Aztec C to give you tremendous programming power and flexibility.

If you have purchased the **Developer Level** of Aztec C for the Amiga, then UniTool utilities and SDB have been included on your Aztec C disks.

If, however, you have purchased the **Professional Level** of Aztec C for the Amiga, it does not include the **make**, **grep**, and **diff** portions of UniTools, even though the documentation for all of these utilities has been included here. However, please note: The **Professional Level** can be easily upgraded to the **Developer Level**. See page 4-4 of your *Aztec C User Guide* for information on how to contact the Update Department.

UniTools includes the utilities **diff**, **grep**, **make**, and **Z editor**. Each of these important utilities is designed to greatly enhance your programming efficiency, as follows:

- The **diff** utility finds the areas of difference in your source files and tells you the changes that must be made to make the files identical.

- The **grep** utility allows you to match patterns in your program and print the locations. This makes it easier for you to either search for, or reference, specific strings and function calls.
- The **make** utility allows you to build large applications without the time-consuming process of using batch files that contain several compile and link command lines. This saves you from having to type command statements over and over.
- The Aztec C **Z editor** is a powerful text editor that allows you to create and edit C source programs. The **Z editor** is similar to the UNIX vi editor.

About This Manual

This manual describes the utilities, **diff**, **grep**, **make**, and **Z**, and gives the commands used in each. It also includes an index of keywords and subjects to make the information in the manual more accessible.

Throughout this manual, we use the following conventions:

input	to indicate data entered by the user(e.g., commands, options, and functions)
DEFINITION	small uppercase bold is used on terms that may be new to the user; most likely will include explanation or definition of term
{choice1 choice2}	braces and a vertical bar mean that you have a choice between two or more items
<i>placeholders</i>	information that must be supplied by the user; for example, <i>filename</i> , <i>range</i> , <i>identifier</i> , etc.
output	to show text that is generated by the computer
[<i>optional</i>]	to show optional information

Technical Support

At Manx, we build dependability into our products. However, if you should find that you need help, rest assured that we provide it.

Refer to your Aztec C Documentation for information on the technical support that Manx provides. If you have any problems with UniTools, follow the procedures outlined in the **Technical Support** chapter so that we may best be able to assist you.

Chapter 2 - Diff

The **diff** utility is designed to display differences between two text files. **diff** will show all differences between *file1* and *file2*, along with information on how to make them similar. **diff** will display the exact lines that are different between the two files, including the relative line numbers in the files.

NAME

diff - Source file comparison utility

SYNOPSIS

diff [-b] *file1 file2*

The Conversion List

diff writes a conversion list to its standard output that describes the changes that need to be made to *file1* to convert it into *file2*. The list is organized into a sequence of items, each of which describes one operation that must be performed on *file1*.

Conversion Items

There are three types of operations that can be specified in a conversion list item:

- adding lines to *file1* from *file2*;
- deleting lines from *file1*;
- replacing (changing) *file1* lines with *file2* lines.

A conversion list item consists of a command line, followed by the lines in the two files that are affected by the item's operation.

The Command Line

An item's command line contains a letter describing the operation to be performed: **a** for adding lines **d** for deleting lines, and **c** for changing lines.

Preceding and following the letter are the numbers of the lines in *file1* and *file2*, respectively, that are affected by the command. If a range of lines in a file is affected, only the beginning and ending line numbers are listed, separated by a comma.

For example, the following command line says to add line 3 of *file2* after line 5 of *file1*:

5a3

and the next command line says to add lines 8, 9, and 10 of *file2* after line 16 of *file1*

16a8, 10

The next command line says to delete lines 100 through 150 from *file1*, and that the last line in *file2* that matched a *file1* line was number 75:

100, 150d75

The following command says to replace (change) line 32 in *file1* with line 33 in *file2*:

32c33

and the next command says to replace lines 453 through 500 in *file1* with lines 490 through 499 in *file2*:

453, 500c490, 499

The Affected Lines

As mentioned above, the lines affected by a conversion item's operation are listed after the item's command line. The affected lines from *file1* are listed first, flagged with a preceding < . Then come the affected lines from *file2*, flagged with a preceding > . The *file1* and *file2* lines are separated by the line

For example, the following conversion item says to add line 6 of *file2* after line 4 of *file1*. Line 6 of *file2* is "for (i=1; i<10;++i)":

```
4a6
> for (i=1; i<10;++i)
```

Since no lines from *file1* are affected by an 'add' conversion item, only the *file2* lines that will be added to *file1* are listed, and the separator line "---" is omitted.

The following conversion item says to delete lines 100 and 101 from *file1*, and that the last *file2* line that matched a *file1* line was numbered 110. The deleted lines were `int a;` and `double b;`. Only the deleted lines are listed, and the separator line "---" is omitted:

```
100,101d110
< int a;
< double b;
```

The following conversion item says to replace lines 53 through 56 in *file1* with lines 60 and 61 in *file2*. Lines 53 through 56 in *file1* are "if (a=b) {", " d = a;", " a++;", and "}". Lines 60 and 61 of *file2* are "if (a==b)" and "d = a++;"

```
53,58c60,61
< if (a=b) {
< d = a;
< a++;
< }
---
> if (a==b)
> d = a++;
```

The -b Option

Option **-b** causes **diff** to ignore trailing blanks (spaces and tabs) and to consider strings of blanks to be identical. If this option is not specified, **diff** considers two lines to be the same only if they match exactly. For example, if **file1** contains the line

```
^abc$
```

(**^** and **\$** stand for "the beginning of the line" and "the end of the line," respectively, and are not actually in the **file**) and if **file2** contains the line

```
^abc  $
```

then **diff** would consider the two lines to be the same or different, depending on whether or not it was started with option **-b**.

And **diff** would consider the lines

```
^a      b c$
```

and

```
^a b c$
```

to be the same or different, depending on whether or not it was started with option **-b**.

diff will never consider blanks to match a null string, regardless of whether **-b** was used or not. So **diff** will never consider the lines

```
^abc$
```

and

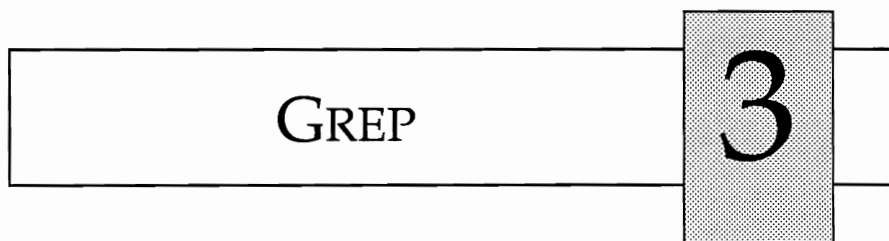
```
^a bc$
```

to be the same.

Differences Between the UNIX and Manx Versions of diff

The Manx and UNIX versions of **diff** are actually most similar when the latter program is invoked with option **-H**. As with the UNIX **diff** when used with option **-H**, the Manx **diff** works best when changed stretches are short and well separated, and works with files of unlimited length.

Unlike the UNIX **diff**, the Manx **diff** does not support option **-E**, **-F**, or **-H**. Unlike the UNIX **diff**, the Manx version requires that both operands be actual files. Because of this, the Manx version of **diff** does not support the features of the UNIX version that allow one operand to be a directory name, (to specify a file in that directory having the same name as the other operand), and that allow one operand to be **'-'** (to specify the **diff** standard input instead of a file).



Chapter 3 - Grep

Description of grep

grep is a program, similar to the UNIX **grep**, that searches for a designated pattern within a specified group of files. The pattern can consist of characters, strings, or a group of strings.

NAME

grep - pattern matching program

SYNOPSIS

grep [*options*] *pattern* [*file1 file2...*]

For example, you could use **grep** to find the function **hello**, as follows:

```
grep hello *.c
```

This command would print out all of the files which reference **hello**.

grep will also allow you to search for patterns in particular places within a file, such as at the beginning of lines or at the ends of lines.

grep will allow you to search multiple files for the pattern you specify. **grep** usually takes two arguments: *pattern*, which indicates what you are looking for, and *file1, file2, etc.* which defines the files you want to search. *file1, file2, etc.* can be the name(s) of a specific file, or can contain wildcard characters. For example, you can say:

```
grep hello *.c
```

or

```
grep hello mike.c
```

or

```
grep hello mike.??1
```

The pattern choices cover a wide range, from simple to complex. For example, you could specify as a *pattern* the letter **m**

```
grep m *.h
```

the function name **hello()**

```
grep hello() *.c
```

or even "the name **hello** but only when it appears at the beginning of a line and with nothing else on that line"

```
grep ^hello$ *.c
```

where **^** means must start at beginning of the line, and **\$** means it is the last item on the line. (See Figure 3.1.)

Typing **grep** on a line all by itself will cause it to search the standard input. Thus, **grep** will return anything you type because it considers your keyboard its source of input.

OPTIONS

The following options are supported by **grep**:

- v Print all lines that do not match the pattern.
- c Print just the name of each file and the number of matching lines that it contained.
- l Print only the names of the files that contain matching lines.
- n Precede each matching line that is printed by its relative line number within the file that contains it.
- f A character in the pattern will match both its upper- and lowercase equivalent.

INPUT FILES

The **files** parameter is a list of files to be searched. If no files are specified, **grep** searches its standard input. Each filename can specify a single file to be searched. A name can also specify a class of files to be searched, using the special characters ***** and **?**. The character ***** matches any string of characters in a filename, and **?** matches any single character. For example,

```
grep int main.c sub1.c sub2.c
```

searches **main.c**, **sub1.c**, and **sub2.c** for the string **int**. The command

```
grep int *.c
```

searches all files whose extension is **c** for the string **int**. The command

```
grep int a*.txt b*.doc
```

searches for the string **int** in each file (1) whose extension is **txt** and *first* character is **a** and (2) whose extension is **.doc** and first character is **b**. The command

```
grep int sub?.c
```

searches for the string `int` in each file whose filename contains four characters, the first three being `sub`, and whose extension is `.c`.

PATTERNS

See Figure 3.1 for a brief synopsis of the special patterns you can use with `grep`.

A pattern consists of a limited form of regular expression. It describes a set of character strings, any of whose members are said to be matched by the regular expression.

Special Patterns Used With `grep`

- `.` matches anything except carriage return or new line
- `[]` matches any one character within the brackets
- `[^]` matches all characters except the ones within the brackets
- `^` matches a pattern only if it is the first item on the line
- `$` matches a pattern only if it is the last item on the line.
- `x*` matches 0 or more occurrences of the character `x`

Note: A backslash, `\`, followed by any of the above characters matches the specified character.

Figure 3.1



Suppose you want to find all lines in the file `prog.c` that contain a four-character string whose first and last characters are `m` and `n`, respectively, and whose other characters you do not care about. The command

```
grep m..n prog.c
```

will do the trick, since the special character `'.'` matches any single character.

➤ []

Suppose that you want to find all lines in the file `file.doc` that begin with a digit. The command

```
grep ^[0123456789] file.doc
```

will do just that. This command can be abbreviated as

```
grep ^[0-9] file.doc
```

And if you wanted to print all lines that do not begin with a digit, you could enter

```
grep ^[^0-9] file.doc
```

➤ \$ and ^

Suppose you want to find the number of the line on which the definition of the function `add` occurs in the file `arith.c`. Entering

```
grep -n add arith.c
```

is not good, because it will print lines in which `add` is called, in addition to the line you are interested in. Assuming that you begin all function definitions at the beginning of a line, you could enter

```
grep -n ^add arith.c
```

to accomplish your purpose.

The character '\$' is a companion to '^', and stands for "the end of the line." So if you want to find all lines in `file.doc` that end in the string `time`, you could enter

```
grep time$ file.doc
```

And the following will find all lines that contain just `.PP`:

```
grep ^.PP$ file.doc
```

► *

Suppose you want to find all lines in the file `prog.c` that contain strings whose first character is `e` and whose last character is `z`. The command

```
grep e.*z prog.c
```

will do that. The `e` matches an `e`, the `*` matches zero or more occurrences of the character that precedes it (in this case a `.` which matches anything) and the `z` matches a `z`.

► \

There are occasions when you want to find the character `'` in a file, and do not want `grep` to consider it to be special. In this case, you can use the backslash character, `'\'`, to turn off the special meaning of the next character.

For example, suppose you want to find all lines containing

```
.PP
```

Entering

```
grep .PP prog.doc
```

is not adequate, because it will find lines such as

```
THE APPLICATION OF
```

since the `'` matches the letter `A`. But if you enter

```
grep \.PP prog.doc
```

`grep` will print only what you want.

The backslash character can be used to turn off the special meaning of any special character. For example,

```
grep '\\n' prog.c
```

finds all lines in `prog.c` containing the string `\n`.

Differences Between the Manx and UNIX Versions of grep

The Manx and UNIX versions of **grep** differ in the options they accept and the patterns they match.

OPTION DIFFERENCES

- Option **-f** is supported only by the Manx **grep**.
- Options **-b** and **-s** are supported only by the UNIX **grep**.

PATTERN DIFFERENCES

Basically, the patterns accepted by the Manx **grep** are a subset of those accepted by the UNIX **grep**.

- The Manx **grep** does not allow a regular expression to be surrounded by **\(** and **\)**.
- The Manx **grep** does not accept the construct **\{m\}**.
- The Manx **grep** does not allow a right bracket, **]**, to be specified within brackets.

Examples

SIMPLE STRING MATCHING

The following command will search the files **file1.txt** and **file2.txt** and print the lines containing the word **heretofore**:

```
grep heretofore file1.txt file2.txt
```

If you are not interested in the specific lines of these files, but just want to know the names of the files containing the word **heretofore**, you could enter

```
grep -l heretofore file1.txt file2.txt
```

The above two examples ignore lines in which **heretofore** contains capital letters, such as when it begins a sentence. The following command will cover this situation:

```
grep -lf heretofore file1.txt file2.txt
```

grep processes all options at once, so multiple options must be specified in one dash parameter. For example, the command

```
grep -l -f heretofore file1.txt file2.txt
```

will not work.

MORE EXAMPLES

If you wish to find all lines in a group of files with a **.c** extension which contain calls to **sprintf** and **fprintf**, but not **printf**, the command

```
grep -n [fs]printf *.c
```

will work.

If, on the other hand, you wish to find only **printf** calls in your **.c** files, and not **sprintf** or **fprintf**, then you can use this **grep** command:

```
grep -n [^a-z]printf *.c
```

Suppose you need to see all calls to the function **Lex**, but only within **if** statements. You would enter

```
grep if.*Lex *.c
```

Finally,

```
grep ^[a-z,A-Z, _][^;)]*)$ *.c
```

will find all function definitions in your **.c** files, assuming that all definitions take place on one line and that a comment is not on the same line as the definition. This is how it works:

1. The **^** states that the following pattern must be the first item on the line.
2. **[a-z,A-Z, _]** states that the first character can only be a letter or a **_**. Preprocessor directive, like **#include** and similar statements, will not fit the pattern.

3. The pattern `[^;)]*` says "do not match until you reach a `;` or a `)` ."
4. The `)$` pattern will only match a `)` at the end of the line. If the previous line matched anything other than a `)`, which could only be an end of line or a `;`, then the whole pattern will fail. In this way, only function definitions which end in a `)` will match.

Chapter 4 - Make

make is a program, similar to the UNIX program of the same name, whose primary function is to create, and keep up-to-date, files that are created from other files, such as programs, libraries, and archives. The **make** program discussed in this chapter is the Manx **make**.

NAME

make - Program maintenance utility

SYNOPSIS

make [*options*] [*name1 name2 ...*]

When told to make a file, **make** first ensures that the files from which the target file is created are up-to-date or current, recreating only the ones that are not. Then, if the target file is not current, **make** creates it.

Interfile dependencies and the commands which must be executed to create files are specified in a file called the makefile, which you must write.

make has a rule-processing capability, which allows it to infer, without being explicitly told, the files on which a file depends and the commands which must be executed to create a file. Some rules are built into **make**; you can define others within the **makefile**.

A rule tells **make** something like this:

"a target file having extension *.x* depends on the file having the same basic name and extension *.y*. To create such a target file, apply the commands"

Rules simplify the task of writing a makefile: A file's dependency information and command sequences need to be explicitly specified in a makefile only if this information cannot be inferred by the application of a rule.

make has a macro capability. A character string can be associated with a macro name; when the macro name is invoked in the **makefile**, it is replaced by its string.

Preview

The rest of this description of **make** is divided into the following sections:

1. The basics
2. Advanced features
3. Examples

Note to Previous Users

One notable change has been implemented in this release of **make**: Default rules, when applied to files in other directories, will place the results in those other directories. Prior to Version 5.0, the results were placed in the current directory.

The Basics

This section presents the basic features of **make**, with which you will be able to start using **make**. The second part of this chapter describes advanced features of **make**.

Before you can begin using **make**, you must know what it does, how to create a simple makefile that contains dependency entries, how to take

advantage of **make**'s rule-processing capability, and, finally, how to tell **make** to create a file. Each of these topics is discussed in the following paragraphs.

What make Does

The main function of **make** is to make a target file "current," where a file is considered "current" if the files on which it depends are current and if it was modified more recently than its prerequisite files. To make a file current, **make** makes the prerequisite files current; then, if the target file is not current, **make** executes the commands associated with the file, which usually recreates the file.

As you can see, **make** is inherently recursive: Making a file current involves making each of its prerequisite files current, making these files current involves making each of their prerequisite files current, and so on.

make is very efficient: it only creates or recreates files that are not current. If a file on which a target file depends is current, **make** leaves it alone. If the target file itself is current, **make** will announce the fact and halt without modifying the target.

It is important to have the time and date set for **make** to behave properly, since it uses the last modified times that are recorded in the 'files' directory entries to decide if a target file is not current.

The Makefile

When **make** starts, it first reads a file, which you must create, called the **makefile**. By default, **make** assumes this file is named **makefile**, but this can be overridden using **make**'s **-f** option. This file contains dependency entries defining interfile dependencies and the commands that must be executed to make a file current. It also contains rule definitions and macro definitions.

In the following paragraphs, we describe only dependency entries. In the "Advanced Features" section of this chapter we discuss the somewhat more advanced topics of rule and macro definition.

A dependency entry in a makefile defines one or more target files, the files on which the targets depend, and the operating system commands that are to be executed when any of the targets is not current. The first line of the entry specifies the target files and the files on which they depend; the line

begins with the target filenames, followed by a colon, followed by one or more spaces or tabs, followed by the names of the prerequisite files.

Please Note: It is important to place spaces or tabs after the colon that separates target and dependent files; on systems that allow colons in filenames, this allows make to distinguish between the two uses of the colon character.

The commands are on the following lines of the dependency information entry. The first character of a command line must be a tab or a space; **make** assumes that the command lines end with the last line not beginning with a tab or space.

For example, consider the following dependency entry:

```
prog: prog.o sub1.o sub2.o
    ln -o prog prog.o sub1.o sub2.o -lc
```

This entry says that the file **prog** depends on the files **prog.o**, **sub1.o**, and **sub2.o**. It also says that if **prog** is not current, **make** should execute the **ln** command. **make** considers **prog** to be current if it exists and if it has been modified more recently than **prog.o**, **sub1.o**, and **sub2.o**.

The above entry describes only the dependence of **prog** on **prog.o**, **sub1.o**, and **sub2.o**. It does not define the files on which the **.o** files depend. For that, we need either additional dependency entries in the makefile or a rule that can be applied to create **.o** files from **.c** files.

For now, we will add dependency entries in the makefile for **prog.o**, **sub1.o**, and **sub2.o**, which will define the files on which the object modules depend and the commands to be executed when an object module is not current. Later we will modify the makefile to make use of **make's** built-in rule for creating a **.o** file from a **.c** file.

Suppose that the **.o** files are created from the C source files **prog.c**, **sub1.c**, and **sub2.c**; that **sub1.c** and **sub2.c** contain a statement to include the file **defs.h**; and that **prog.c** does not contain any **#include**

statements. Then the following long-winded makefile could be used to explicitly define all the information needed to create **prog**

```
prog: prog.o sub1.o sub2.o
    ln -o prog prog.o sub1.o sub2.o -lc
prog.o: prog.c
    cc prog.c
sub1.o: sub1.c defs.h
    cc sub1.c
sub2.o: sub2.c defs.h
    cc sub2.c
```

This makefile contains four dependency entries: for **prog**, **prog.o**, **sub1.o**, and **sub2.o**. Each entry defines the files on which its target file depends and the commands to be executed when its target is not current. The order of the dependency entries in the makefile is not important.

We can use this makefile to make any of the four target files defined in it. If none of the target files exists, and if the name of the makefile is **makefile**, then entering

```
make prog
```

causes **make** to compile and assemble all three object modules from their C source files, and then create **prog** by linking the object modules together.

Suppose that you create **prog** and then modify **sub1.c**. Then telling **make** to make **prog** causes **make** to compile and assemble **sub1.c** only, and then recreate **prog**.

If you then modify **defs.h**, and tell **make** to create **prog**, **make** will compile and assemble **sub1.c** and **sub2.c**, and then recreate **prog**.

You can tell **make** to make any file defined as a target in a dependency entry. Thus, if you want to make **sub2.o** current, you could enter

```
make sub2.o
```

A makefile can contain dependency entries for unrelated files. For example, the following dependency entries can be added to the above makefile:

```
hello: hello.o
    ln hello.o -lc
hello.o: hello.c
    cc hello.c
```

With these dependency entries, you can tell **make** to make **hello** and **hello.o**, in addition to **prog** and its object files.

RULES

You can see that the makefile describing a program built from many **.o** files would be huge if it had to explicitly state that each **.o** file depends on its **.c** source file and is made current by compiling its source file.

This is where rules are useful. When a rule can be applied to a file that **make** has been told to make or that is a direct or indirect prerequisite of it, the rule allows **make** to infer, without being explicitly told, the name of a file on which the target file depends and/or the commands that must be executed to make it current. This in turn allows makefiles to be very compact, only specifying information that **make** cannot infer by the application of a rule.

Some rules are built into **make**; you can define others in a makefile. In the rest of this section, we describe the properties of rules and how you write makefiles that make use of **make**'s built-in rule for creating a **.o** file from a **.c** file.

make's Use of Rules

A rule specifies a target extension, source extension, and sequence of commands. Given a file that **make** wants to make, it searches the rules known to it for one that meets the following conditions:

- The rule's target extension is the same as the file's extension;
- A file exists that has the same basic name as the file **make** is working on and that has the rule's source extension.

If a rule is found that meets these conditions, **make** applies the first such rule to the file it is working on, as follows:

- The file having the source extension is defined to be a prerequisite of the file with the target extension;
- If the file having the target extension does not have a command sequence associated with it, the rule's commands are defined to be the ones that will make the file current.

One rule built into **make**, for converting **.c** files into **.o** files, says

"a file having extension **.o** depends on the file having the same basic name, with extension **.c**. To make current such a **.o** file, execute the command

```
cc x.c
```

where *x* is the name of the file."

Another built-in rule exists for converting **.asm** files into **.o** files, using the Manx assembler.

Example

The **.c** to **.o** rule allows us to abbreviate the long-winded makefile shown above as follows:

```
prog: prog.o sub1.o sub2.o
    ln -o prog prog.o sub1.o sub2.o -lc
sub1.o sub2.o: defs.h
```

In this abbreviated makefile, a dependency entry for **prog.o** is not needed; using the built-in **.c-to-.o** rule, **make** infers that the **prog.o** depends on **prog.c** and that the command **cc prog.c** will make **prog.o** current.

The abbreviated makefile says that both **sub1.o** and **sub2.o** depend on **defs.h**. It does not say that they also depend on **sub1.c** and **sub2.c**, respectively, or that the compiler must be run to make them current; **make** infers this information from the **.c** to **.o** rule. The only information given in the dependency entry is that which **make** could not infer by itself: that the two object files depend on **defs.h**.

Interaction of Rules and Dependency Entries

As we showed in the above example, a rule allows you to leave some dependency information unspecified in a makefile. The **prog.o** entry in the long-winded makefile shown earlier in this section was not needed, because its information could be inferred by the **.c to .o** rule. And the dependence of **sub1.o** and **sub2.o** on their respective C source files, and the commands needed to create the object files was also not needed, since the information could be inferred from the **.c to .o** rule.

There are occasions when you do not want a rule to be applied; in this case, information specified in a dependency entry will override that which would be inferred from a rule. For example, the following dependency entry in a makefile

```
add.o:
    cc -DFLOAT add.c
```

causes **add.o** to be compiled using the specified command rather than the command specified by the **.c to .o** rule. **make** still infers the dependence of **add.o** on **add.c**, using the **.c to .o** rule, however.

Advanced Features

The last section presented the basic features of **make** to help you begin using **make**. This section presents the rest of **make's** features.

Dependent Files

make supports dependencies on files in directories other than the current directory. This is best illustrated by the following example:

Suppose the current directory is **/src1** and that the program is being built from source there and in the **/src2** directory. The makefile line would look like this:

```
program: /src/main.o /src2/sub.o
    ln -o program /src1/main.o /src2/sub.o -lc
```

This builds **main.o** and **sub.o** from their sources, putting the object files (the **.o's**) in the same directory as the sources. The **.o** file will always be placed in the same directory as the source file.

Note: Unfortunately, there is no way to say "all my .o's are in one directory and all my .c's are in another."

Any further references to these files should be with the same path or else **make** will be confused.

For object files to be dependent on files in another directory, the full pathname must be used as in:

```
/src2/sub.o:/header/defs.h
```

There are a few conventions that you must be aware of when naming your files. These are as follows:

- If the filename contains a colon (for example, because the filename defines the volume on which the file is located), the colon must be followed by characters other than spaces or tabs, so that **make** can distinguish between this use of the colon character and its use as a separator between the target and dependent files in a dependency line. This should not be a problem, since most systems do not allow filenames to contain spaces or tabs.
- All references to a file must use the same name. For example, if a file is referred to in one place using the name

```
/root/src/foo.c
```

then all references to the file must use this exact same name. The name

```
../src/foo.c
```

would not match.

MACROS

make has a simple macro capability that allows character strings to be associated with a macro name and to be represented in the makefile by the name. The following paragraphs describe how to use macros within a makefile, then how they are defined, and finally some special features of macros.

Using Macros

Within a makefile, a macro is invoked by preceding its name with a dollar sign; macro names longer than one character must be parenthesized. For example, the following are valid macro invocations:

```
$(CFLAGS)
$2
$(X)
$X
```

The last two invocations are identical.

When **make** encounters a macro invocation in a dependency line or command line of a makefile, it replaces it with the character string associated with the macro. For example, suppose that the macro **OBJECTS** is associated with the string **a.o b.o c.o d.o**. Then the dependency entries:

```
prog: prog.o a.o b.o c.o d.o
    ln prog.o a.o b.o c.o d.o
a.o b.o c.o d.o: defs.h
```

within a makefile could be abbreviated as:

```
prog: prog.o $(OBJECTS)
    ln prog.o $(OBJECTS)
$(OBJECTS): defs.h
```

Defining Macros in a makefile

A macro is defined in a makefile by a line consisting of the macro name, followed by the character **=**, followed by the character string to be associated with the macro.

For example, the macro **OBJECTS**, used above, could be defined in the makefile by the line

```
OBJECTS = a.o b.o c.o d.o
```

A makefile can contain any number of macro definition entries. A macro definition must appear in the makefile before the lines in which it is used.

Defining Macros in a Command Line

A macro can be defined in the command line that starts **make**. The syntax for a command line definition has the following form:

```
make MACRO=text
```

For example, the following command assigns the value **-DFLOAT** to the macro **CFLAGS**:

```
make CFLAGS=-DFLOAT
```

Another example would be as follows:

```
make MACRO=text
```

Note that the equal sign (=) is the key to having it be a macro definition. If the macro is to be empty, enter:

```
make MACRO=
```

Command line macro definition always overrides makefile macro definition. Be careful about your macro definitions if you are using **make** on systems other than the Amiga. Some operating systems do not support quotes around text, so any white space in the macro text causes the macro definition to terminate prematurely.

Macros Used by Built-In Rules

make has two macros, **CFLAGS** and **AFLAGS**, that are used by the built-in rules. These macros by default are assigned the null string. This can be overridden by a macro definition entry in the makefile.

For example, the following would cause **CFLAGS** to be assigned the string **-T**:

```
CFLAGS = -T
```

These macros are discussed below in the description of built-in rules.

Special macros

There are three special macros: **\$\$**, **\$***, and **\$@**. **\$\$** represents the dollar sign. The other two are discussed below.

Before issuing any command, two special macros are set: `$@` is assigned the full name of the target file to be made, and `$*` is the name of the target file, without its extension. Unlike other macros, these can only be used in command lines, not in dependency lines.

For example, suppose that the files `x.c`, `y.c`, and `z.c` need to be compiled using the option `-DFLOAT`. The following dependency entry could be used:

```
x.o y.o z.o:
cc -DFLOAT $*.c
```

When `make` decides that `x.o` needs to be recreated from `x.c`, it will assign `$*` to the string `x`, and the command

```
cc -DFLOAT x.c
```

will be executed. Similarly, when `y.o` or `z.o` is made, the command

```
cc -DFLOAT y.c
```

or

```
cc -DFLOAT z.c
```

will be executed.

The special macros can also be used in command lines associated with rules. In fact, the `$@` macro is primarily used by rules. We will discuss this more in the **Rules** section that follows.

RULES

Earlier we presented the basic features of rules: what they are and how they are used. We noted that rules could be defined in a makefile and that some rules are built into `make`. In the following paragraphs, we describe how rules are defined in a makefile and list the built-in rules.

Rule Definition

A rule consists of a source extension, target extension, and command list. In a makefile, an entry defining a rule consists of a line defining the two extensions, followed by lines containing the commands.

The line defining the extensions consists of the source extension, immediately followed by the target extension, followed by a colon.

All command lines associated with a rule must begin with a tab or space character. The first line following the extension line that does not begin with a tab or space terminates the commands for the rule.

For example, the following rule defines how to create a file having extension `.rel` from one having extension `.c`:

```
.c.rel:
    cc -o $@ $*.c
```

The first line declares that the rule's source and target extension are `.c` and `.rel`, respectively.

The second line, which must begin with a tab, is the command to be executed when a `.rel` file is to be created using the rule.

Note the existence of the special macros `$@` and `$*` in the command line. Before the command is executed to create a `.rel` target file using the rule, the macro `$@` is replaced by the full name of the target file, and the macro `$*` by the name of the target, less its extension.

Thus, if `make` decides that the file `x.rel` needs to be created using this rule, it will issue the command

```
cc -o x.rel x.c
```

If a rule defined in a makefile has the same source and target extensions as a built-in rule, the commands associated with the makefile version of the rule replace those of the built-in version. For example, the built-in rule for creating a `.o` file from a `.c` file looks like this:

```
.c.o:
    cc $(CFLAGS) -o $@ $*.c
```

If you want the rule to generate an assembly language listing, include the following rule in your makefile:

```
.c.o:
    cc $(CFLAGS) -a $*.c
    as -ZAP -l -o $@ $*.asm
```

Built-in Rules

The following rules are built into **make**. The order of the rules is important, since **make** searches the list beginning with the first one, and applies the first applicable rule that it finds.

```
.c.o:
    cc $(CFLAGS) -o $@ $*.c
.asm.o:
    as $(AFLAGS) -o $@ $*.asm.
```

Thus, if both a **.asm** and a **.c** file exist with the same basic name, the **.c** file would be used and not the **.asm** file

The two macros **CFLAGS** and **AFLAGS** that are used in the built-in rules are built into **make**, having the null character string as their values. To have **make** use other options when applying one of the built-in rules, you can define the macro in the makefile.

For example, if you want the options **-t** and **-DDEBUG** to be used when **make** applies the **.c** to **.o** rule, you can include the line

```
CFLAGS = -T -DDEBUG
```

in the makefile. Another way to accomplish the same result is to redefine the **.c** to **.o** rule in the makefile; this, however, would use more lines in the makefile than the macro redefinition.

COMMANDS

In this section we want to discuss the execution of operating system commands by **make**.

Allowed Commands

A command line in a dependency entry or rule within a makefile can specify any command that you can enter at the keyboard.

This includes batch commands, commands built into the operating system, and commands that cause a program to be loaded and executed from a disk file.

Logging Commands and Aborting make

Normally, before **make** executes a command, it writes the command to its standard output device; and when the command terminates, **make** halts if the command's return code was non-zero. Either or both of these actions can be suppressed for a command, by preceding the command in the makefile with a special character:

- @ Tells **make** not to log the command;
- Tells **make** to ignore the command's return code.

For example, consider the following dependency entry in a makefile:

```
prog: a.o b.o c.o d.o
    ln -o prog a.o b.o c.o d.o -lc
@echo "All done!"
```

when the **echo** command is executed, the command itself will not be logged to the console.

Long Command Lines

Makefile commands that start a Unix program, such as **cc**, **as**, or **ln**, or that start a program created with **cc**, **as**, **ln**, and **c.lib**, can specify a command line containing up to 2048 characters.

For example, if a program depends on fifty modules, you could associate them with the macro **OBJECTS** in the makefile, and also include the dependency entry

```
prog: $(OBJECTS)
    ln -o prog $(OBJECTS) -lc
```

This will result in a very long command line being passed to **ln**.

For the execution of other commands, the command line can contain at most 127 characters.

Makefile Syntax

This section has already presented most of the syntax of a makefile; that is, how to define rules, macros, and dependencies. However, we must discuss

two features of a makefile syntax not presented elsewhere: comments and line continuation.

Comments

make assumes that any line in a makefile whose first character is '#' is a comment, and ignores it. For example:

```
#
# the following rule generates
# an 8080 object module
# from a C source file:
#
.c.o80:
    cc80 -o cc.tmp $*.c
    as80 -ZAP -o $*.o80 cc.tmp
```

Line Continuation

Many of the items in a makefile must be on a single line: A macro definition, the file dependency information in a dependency entry, and a command that **make** is to execute must each be on a single line.

You can tell **make** that several makefile lines should be considered to be a single line by terminating each of the lines, except the last, with the backslash character, '\'. When **make** sees this, it replaces the current line's backslash and newline, and the next line's leading blanks and tabs by a single blank, thus effectively joining the lines together.

The maximum length of a makefile line after joining continued lines is 2048 characters.

For example, the following macro definition equates **OBJ** to a string consisting of all the specified object module names.

```
OBJ = printf.o fprintf.o format.o\  
scanf.o fscanf.o scan.o\  
getchar.o getc.o
```


As another example, the following dependency entry defines the dependence of `driver.lib` on several object modules, and specifies the command for making `driver.lib`:

```
driver.lib: driver.o printer.o \  
    in.o \33  
    out.o  
lib -o driver.lib driver.o  
    printer.o \  
    in.o out.o
```

This second example could have been more cleanly expressed using a macro:

```
DRIVOBJ= driver.o printer.o\  
    in.o out.o  
driver.lib: $(DRIVOBJ)  
lib -o driver.lib $(DRIVOBJ)
```

This was done to show that dependency lines and command lines can be continued, too.

Starting make

We have already discussed how `make` is told to make a single file. Entering

```
make filename
```

makes the file named `filename`, which must be described by a dependency entry in the makefile. And entering

```
make
```

makes the first file listed as a target file in the first dependency entry in the makefile.

In both of these cases, `make` assumes the makefile is named `makefile` and that it is in the current directory on the default drive.

In this section we want to describe the other features available when starting `make`.

The Command Line

The complete syntax of the command line that starts **make** is:

make [-n] [-f *makefile*] [-a] [macro=*str*] [*file1*] [*file2*] ...

Square brackets indicate that the enclosed parameter is optional.

The parameters *file1*, *file2* ... are the names of the files to be made. Each file must be described in a dependency entry in the makefile. They are made in the order listed on the command line.

The other command line parameters are options and can be entered in upper- or lowercase. Their meanings are:

- | | |
|---------------------------|---|
| -n | Suppresses command execution. make logs the commands it would execute to its standard output device, but does not execute them. |
| -f <i>makefile</i> | The name of the makefile is <i>makefile</i> . |
| -a | Forces make to make all files upon which the specified target files directly or indirectly depend, and to make the target files, even those that it considers current. |
| MACRO=<i>str</i> | Creates a macro named MACRO , and assigns <i>str</i> as its value. |

make's Standard Output

make only uses its standard output device for printing error messages and for logging commands. You can redirect **make**'s standard output device in the normal way.

The standard input and output devices of a program started by **make** inherit the standard input and output of the make program, unless the command that started the program explicitly redirected one or both of them.

Differences between the Manx and UNIX make Programs

The Manx **make** supports a subset of the features of the UNIX **make**. The following comments present features of the UNIX **make** that are not supported by the Manx **make**.

- The UNIX **make** will let you make a file that is not defined as a target in a makefile dependency entry, so long as a rule can be applied to create it. The Manx **make** does not allow this. For example, if you want to create the file **hello.o** from the file **hello.c** you could say, on UNIX

```
make hello.o
```

even if **hello.o** was not defined to be a target in a makefile dependency entry. With the Manx **make**, you would have to have a dependency entry in a makefile that defines **hello.o** as a target.

- The UNIX **make** supports the following options, which are not supported by the Manx **make**:

p, i, k, s, r, b, e, m, t, d, q

- The Manx **make** supports the option **-a**, which is not supported by the UNIX **make**.
- The special names **DEFAULT**, **.PRECIOUS**, **.SILENT**, and **.IGNORE** are supported only by the UNIX **make**.
- Only the UNIX **make** allows the makefile to be read from **make's** standard input.
- Only the UNIX **make** supports the special macros **\$?** and **\$%** and allows an uppercase **D** or **F** to be appended to the special macros, which thus modifies the meaning of the macro.
- Only the UNIX **make** requires that the suffixes for additional rules be defined in a **.SUFFIXES** statement.
- Only the UNIX **make** allows a target to depend on a member of a library or archive.

Examples

This example shows a makefile for making several programs. Note the entry for `arc`. This does not result in the generation of a file called `arc`; it is only used so that `arcv` and `mkarcv` can be generated by entering `make arc`.

```
#
# rules:
#
.c.o80:
    cc80 -DTINY -o $@ $*.c
#
# macros:
#
OBJ=make.o parse.o scandir.o \
dumptree.o rules.o command.o
#
# dependency entry for making make:
make: $(OBJ)
    ln -o make $(OBJ) -lc
# dependency entries for making arcv & mkarcv:
#
arc: mkarcv arcv
mkarcv: mkarcv.o
    ln -o mkarcv mkarcv.o -lc
arcv: arcv.o
    ln -o arcv arcv.o -lc
#
# dependency entries for making
# CP/M-80 versions of arcv & mkarcv:
#
mkarcv80.com: mkarcv.o80
    ln80 -o mkarcv80.com mkarcv.o80 -lt -lc
arcv80.com: arcv.o80
    ln80 -o arcv80.com arcv.o80 -lt -lc
$(OBJ): libc.h makefile
```

The next example uses **make** to make a library, **my.lib**. Three directories are involved: the directory **libc** and two of its subdirectories, **sys** and **misc**. The C and assembly language source files are in the two subdirectories. There are makefiles named **makefile** in each of the three directories, and this example makes use of all of them. With the current directory being **libc**, you enter

```
make my.lib
```

This starts **make**, which reads the makefile in the **libc** directory. **make** will change the current directory to **sys** and then start another **make** program.

This second **make** compiles and assembles all the source files in the **sys** directory, using the makefile that is in the **sys** directory.

When the **sys make** finishes, the **libc make** regains control, and then starts yet another **make**, which compiles and assembles all the source files in the **misc** subdirectory, using the makefile that is in the **misc** directory.

When the **misc make** is done, the **libc make** regains control and builds **my.lib**. You can then remove the object files in the subdirectories by entering

```
make clean
```

The following files contain the makefiles for this example:

The makefile In the 'libc' Directory

```
my.lib: sys.mk misc.mk  
rm my.lib  
libutil -o my.lib -f my.bld
```

```
sys.mk:
    cd sys
    make
    cd ..
misc.mk:
    cd misc
    make
    cd ..
clean:
    cd sys
    make clean
    cd ..
    cd misc
    make clean
    cd ..
```

Makefile for the 'sys' Directory

```
REL=asctime.o bdos.o begin.o chmod.o \
    croot.o csread.o ctime.o \
dostime.o dup.o exec.o execl.o execlp.o \
    execlv.o execvp.o \
fexec.o fexecl.o fexecv.o ftime.o \
    getcwd.o getenv.o \
isatty.o localtime.o mkdir.o open.o stat.o \
    system.o time.o \
    utime.o wait.o dioctl.o ttyio.o access.o
syserr.o \
```

```
COPT=
HEADER=../header
.c.o:
    cc $(COPT) -I$(HEADER) *.c -o $@
    sqz $@
.asm.o:
    as *.asm -o $@
    sqz $@
all: $(REL)
clean:
    rm *.o
```

Makefile for the 'misc' Directory

```
REL= atoi.o atol.o calloc.o ctype.o format.o \
    malloc.o \
    qsort.o sprintf.o sscanf.o fformat.o fscan.o
COPT=
HEADER=../header
.c.o:
    cc $(COPT) -I$(HEADER) *.c -o $@
    sqz $@
.asm.o:
    as *.asm -o $@
    sqz $@
all: $(REL)
fformat.o: format.c
    cc -I$(HEADER) -DFLOAT format.c -o fformat.o
fscan.o: scan.c
    cc -I$(HEADER) -DFLOAT scan.c -o fscan.o
clean:
    rm *.o
```


Chapter 5 - Z Editor

Z is a text editor for creating source programs, usually in the C programming language. Z has the following features:

- Similarity to the UNIX editor Vi: If you know Vi, you know Z.
- Full-screen editor: The screen acts as a window into the file being edited.
- A wealth of commands, specified with only a few keystrokes, that allow you to edit quickly and efficiently. The simple and natural way of entering commands and the mnemonic assignment of commands to keys make the commands easy to remember and use.
- An interface to the Manx **QuikFix** facility. **QuikFix** launches the editor directly from the compiler, loads the source file, positions to the first statement in error, and displays the error message. You can then correct the error and automatically go to the next error. After you correct the errors, you enter, :wq, the changes are saved, and the compiler is re-launched. If necessary, the process repeats. **QuikFix** cuts out administrative time and eliminates the bother of matching up error messages and source lines. There is no fumbling around, there are no annoying distractions. **QuikFix** orchestrates the motions and lets you focus on the solutions.
- Commands for the following:

- ☐ Bringing different sections of a file into view
 - ☐ Inserting text
 - ☐ Making changes to text
 - ☐ Rearranging text by moving blocks of text around and by inserting text from other files
 - ☐ Accessing files
 - ☐ Searching for character strings and "regular expressions"
- Several commands that are useful for editing C programs, including commands for finding matching parentheses, square brackets, and curly braces; for finding the beginning of the next or preceding function; and for finding the next or preceding blank line.
 - Most commands can be easily executed repeatedly.
 - Sequences of commands, called macros, can be defined and executed one or more times.
 - Changes are made to an in-memory copy of a file; the file itself is not changed until a command is explicitly given.
 - Editing feature that allows the operator to request that a file containing a certain function be edited—**Z** finds the file and prepares it for editing.

Requirements

The maximum file size that you may edit using **Z** must not exceed your system's total memory size, *minus 64K*. In other words, to use **Z** editor, you must have at least 64K of available memory above and beyond the file you want to edit.

As you begin editing, **Z** will initially allocate a 16K edit buffer; **Z** will allocate additional 16K buffers when necessary.

Components

The Z package contains two programs:

- **Z** - the text editor
- **ctags**, - a utility for creating a file that relates tags to copies of C source files

Getting Started

Z is a very powerful tool for creating and editing C source programs, but its wealth of commands and options can be overwhelming to someone not familiar with it. This chapter gets you using Z as quickly as possible by presenting a small subset of the Z commands with which you can create and edit programs. Then, with the ability to create and edit programs, you can continue reading the rest of this chapter at your leisure to learn about the other features and commands of Z.

The first part of this chapter describes how to create a new C program, and the second part, how to edit an existing program.

CREATING A NEW PROGRAM

You start Z by entering:

```
z hello.c
```

where **hello.c** represents the file to be edited. Since we are creating a new program, the file does not yet exist, therefore, Z displays a message on its status line (which may be either the first or last line of the display, depending on the system on which Z is running). On systems that use the first display line for status information, the screen looks like this:

```
"hello.c" No such file or directory
...
```

with the cursor on the left-hand column of the second line. On systems that use the last display line for status information, the screen looks like this:

```
...
"hello.c" No such file or directory
```

with the cursor on the left-hand column of the first line.

Z is now waiting for you to enter a command.

The Screen

As mentioned above, **Z** uses the one line of the display for displaying information and for echoing the characters of some commands that are entered.

The rest of the lines on the screen display the text of the file being edited.

The tilde characters on the screen lines tell you that **Z** has reached the end-of-file. These characters are not actually in the file.

Modes of Z

Z has two modes: command and insert, that allow you to enter commands and to insert text, respectively.

Insert Mode

With **Z** in insert mode, characters that you type are entered into a memory-resident buffer; the characters do not appear in the file until you exit insert mode and explicitly issue a command that causes **Z** to write the buffer to the file.

Z has several commands for entering insert mode; the one we want to use, **i**, allows text to be entered before the cursor. Type **i**. Notice that **Z** does not echo this command on the screen; it only does that for a few commands. Notice also that you are in insert mode, as evidenced by the message

<INSERT>

on the right-hand side of the status line.

You may now enter a program, just as you would on a typewriter. Notice that the cursor is positioned where the next character will be entered. Enter the "hello world" program:

```
main()
{
printf("hello, world\n");
}
```

When you press the <CR> key after entering the `printf` line, the cursor was left positioned on the next line of the screen underneath the first nonwhite space character of the preceding line. This feature, "autoindent," is useful when creating C programs, encouraging statements within a compound statement to be indented and lined up. Autoindent can be disabled and enabled, and we will show you how later.

We want the closing curly brace of the main function to be on the first column of the line, not indented. So after you type the semicolon and then return at the end of the `printf` line, type the backspace key to get back to the first column, and then type the `)` key.

The backspace key can also be used to backspace over characters that you incorrectly type.

During insert mode, if you hold down the control key and type a `W`, the previous word typed is deleted.

When you have finished inserting the program, hit the escape key to exit insert mode and return to command mode. The key used as the escape key varies from system to system.

Exiting Z

To write the program you have just entered from the text buffer to the disk file `hello.c` and then exit `Z`, type `ZZ`. (Note that the "ZZ" must be typed as *uppercase*; the entry "zz" will not work.)

Occasionally you may want to exit `Z` without writing the text you have entered to a file; in this case, you would type

`:q!`

followed by a carriage return <CR>.

EDITING AN EXISTING FILE

The following describes the commands you need to make changes to an existing file.

Starting and Stopping Z

You get in and out of **Z** when editing an existing file just as you do when creating a new file. To start **Z**, enter

```
z hello.c
```

where **hello.c** is the name of the file to be edited.

Z reads the specified file into the text buffer, displays the first screen of file text, displays the file's statistics (name, number of lines, number of characters) on the status line, positions the cursor at the first character of the first line, and enters command mode, waiting for you to enter a command.

To stop **Z** and save the changes you have made, put **Z** in command mode and enter:

```
ZZ
```

(Again, note that the "ZZ" must be typed as *uppercase*; the entry "zz" will not work.) **Z** knows if you made changes to the original text or not; if you did, **Z** saves the original file by changing the extension of its name to .bak and then writes the modified text to a new file having the specified name. If a .bak file with that name already exists it will be deleted before the rename occurs.

If you did not make any changes, the **ZZ** command causes **Z** to halt without changing any disk files.

The command **:q!** **QUITS Z** without writing anything to the file being edited.

Command Line Options

With Aztec C version 5, **z** supports the two following options to be used in conjunction with **QuikFix**.

- e The option **-e** when **z** is invoked causes **z** to parse the **AztecC.Err** file generated by the compiler. **z** reads the first line of the file and determines the name of the file in which the errors occurred. It then reads and stores all the errors associated with the file up to 25 errors. Additional errors for this file are left in the **AztecC.Err** file for later parsing. The lines read are removed from the file. If all are read, the file is removed. **z** then reads the target

source file and positions the cursor on the correct line and column and displays the appropriate error or warning message.

- o The option -o causes z when run in the background to open a window at the specified position with the specified size and title. This string is appended to the "RAW:" which is used to open the window and follows the standard Amiga conventions for console windows.

The Cursor

Before describing the commands for viewing and changing the text in Z's memory-resident buffer, we need to discuss the cursor.

In Z, the character position in the text that is pointed to by the cursor acts as a reference point: Most commands perform an action relative to that position. For example, the i command, described in the last section, allows you to enter text before the cursor. And the x command, to be discussed, deletes the character at which the cursor is located.

Therefore, we describe two types of commands in this chapter: those that move the cursor around in the text, thus bringing different sections of text into view, and those that modify text in the vicinity of the cursor.

Moving Around in the Text: Scrolling

The text you created for the "hello, world" program easily fit on a single screen. But most text files are too large to be viewed all at once, so we need commands to bring different sections into view.

Two such commands are the "scroll" commands: "scroll down," represented by the character Control-D, and "scroll up," represented by Control-U. That is, to execute the "scroll down" command, you hold down the control key and then press the D key. The rest of this manual refers to control characters using notation of the form ^D rather than Control-D, for brevity. Thus, the scroll up and scroll down commands are represented as ^U and ^D, respectively.

A scroll command moves the screen up or down in the file, bringing another half-screen of text into view. It is as if the text were on a reel of tape and the

screen is a viewer: Scrolling down moves the viewer down the reel, and scrolling up moves the viewer up the reel.

When scrolling, the cursor will be left on the same position within the text after the scroll as before, if that position is still within view. Otherwise, the cursor is moved to a line in the text which was newly brought into view.

Moving Around In the Text: the Go Command

Scrolling is one way to move around in the text, but it is slow. If we have a large text file to which we want to append text, it would take a long time and many scroll commands to reach the end.

The **go** command, **g**, is one way to move rapidly to the point of interest in the text: Entering **g** by itself will move the cursor to the end of the text and, if necessary, redraw the screen with the text which precedes it.

The **g** command can also be preceded by the number of the line of interest; the cursor will move to the beginning of that line. So to move back to the first line of text, enter:

1g

The **g** command can be used to move to any line within the text, but since you usually do not know the numbers of the lines, the **g** command is mainly used to move to the beginning and end of the text.

Moving Around In the Text: String Searching

So, scrolling allows us to take a casual stroll through text, and the **g** command to move rapidly to the beginning and end of the file. What we need is a command to rapidly move to a specific point in the middle of the text.

The string search command, **/**, is such a command. When you enter **/**, followed by the string of interest, followed by a carriage return, **Z** searches forward in the text from the cursor position, looking for the string. If **Z** reaches the end of the text without finding the string, it will wrap around and continue searching from the beginning of the text.

If the string is found, the cursor is positioned at its first character and, if necessary, the screen is redrawn with its surrounding text.

If the string is not found, a message saying so is displayed on the status line of the screen and the cursor is not moved.

While the string search command and its string are being entered, the characters are displayed on the status line, and normal editing operations can be used, such as backspacing over mistyped characters.

Z remembers the last string searched for. To repeat the search, enter the find next string command, **n**.

Finely Tuned Moves

With the commands presented up to now, you can move to the area of interest in the text. The next few paragraphs present commands that move the cursor from somewhere within the area of interest to a specific character position from which changes will be made.

Some commands for this, from the many available in Z, are:

- | | |
|----------------------|---|
| - | Moves the cursor up one line, to the first non-whitespace character on the line. |
| CR (carriage return) | Moves the cursor down one line, to the first non-whitespace character on the line. |
| space | Moves the cursor right on the line on which the cursor is located. |
| backspace | Moves the cursor left on the line on which the cursor is located. |

These commands can be preceded by a number, which cause the command to be performed the specified number of times. For example,

- | | |
|----------|---|
| 3- | Moves the cursor up three lines. |
| 5<space> | Moves the cursor right five characters. Note that <space> represents the space bar. |

Deleting Text

You now have a repertoire of commands that allows you to move the cursor fairly quickly to any location in a text file. We are ready to move on to a few commands for modifying the text.

Two such commands for deleting text are **DELETE CHARACTER**, **x**, and **DELETE LINE**, **dd**:

- x** Deletes the character under the cursor.
- dd** Deletes the entire line on which the cursor is located.

Each of these commands can be preceded by a number, causing the command to be repeated the specified number of times. For example,

- 2x** Deletes two characters.
- 3dd** Deletes three lines.

More Insert Commands

You already know one command for inserting text: **i**, which allows text to be inserted before the cursor. We need a few more insert commands:

- a** Enters insert mode such that text is inserted following the cursor.
- (*Lowercase o*) Creates a blank line below the current line (i.e., the line on which the cursor is located), moves the cursor to the new line, and enters insert mode.
- (*Uppercase o*) Same as **o**, but the new line is above the current line.

Summary

With the set of commands presented in this chapter, you can edit any text file. You should continue reading this chapter to learn more about **Z**, while you use the basic command set for performing your editing chores.

You will find that **Z** has many more capabilities that allow you to perform functions more quickly and with fewer keystrokes than with the basic command set, and that allow you to perform functions that you cannot perform with the basic command set.

More Commands

This section describes the rest of the features and commands of **Z**, building and expanding on the information presented in the previous section.

THE SCREEN

We have already discussed the basic details on **Z**'s use of the screen. Some of the more complex details concerning **Z**'s use of the screen are addressed below.

Displaying Unprintable Characters

A file edited by **Z** can contain any character whose ASCII value in decimal is less than 128, including unprintable characters, such as SOH (start of heading), LF (line feed), and ESC (escape). **Z** displays unprintable characters as two characters; the first is ^, and the second is the character whose ASCII value equals that of the character itself plus 0x40. For example, the unprintable character SOH is displayed as the pair of characters ^A, since the ASCII value of SOH is 1, and 1 plus 0x40 is 0x41, which is the ASCII value for the character "A".

Displaying Lines That Do Not Fit on the Screen

We have already stated that lines beyond the end of the file are displayed with the character ~ in the first column of the line on the screen. When you see the ~ character in the leftmost column of a line on the screen, this usually signifies that this line of the display does not contain a line of text. But lines that do not fit on the screen are displayed by **Z** in a similar manner.

Z allows lines to be entered that are longer than a screen line. Normally, **Z** simply displays such lines on several screen lines. In some cases, however, the entire line will not fit on the screen. For example, if the cursor is positioned at the beginning of the file, it may not be possible to display the text of an overly-large line at the bottom of the screen. In this case, **Z** displays an @ character in the first column of the screen lines on which the text would be displayed.

Thus, when you see the @ character in the leftmost column of a line on the screen, this signifies that the text that would have appeared on this line of the screen was too big, and not that the @ character is in the text.

Commands

When most commands are entered, **Z** does not echo the characters on the screen. However, two commands for which **Z** does display the characters on the screen are (1) those commands whose first character begins with ":" and (2) the string search commands.

For these commands, the characters are displayed on the screen status line, and can be backspaced over and reentered, if necessary. Also, **Z** does not act on such commands until you type the carriage return key, CR.

Special Keys

There are two keys that have special meaning for **Z**: the escape key, which is used to exit insert mode, and the control key, which is used in conjunction with another key to generate control characters. The actual keys used for these functions vary from system to system.

PAGING AND SCROLLING

Previously, we described commands for scrolling through text, **^U** and **^D**. Another pair of commands allow you to page, instead of scroll, through text. The commands are **^B** and **^F**, which page backwards and forwards, respectively.

A page command brings the previous or next screen of text into view by redrawing the screen with the new text. Whereas scrolling was described as a viewer moving over a reel of tape, paging can be described as turning the pages of a book.

Paging moves you through text more quickly than scrolling does. However, since paging redraws the screen all at once, while scrolling changes it gradually, it is often more difficult to keep a sense of continuity when paging than when scrolling. As an aid to continuity when paging, two lines of text which were previously in view are still in view after paging.

Scroll commands can be preceded by a value specifying the number of lines to be scrolled up or down. If a number is not specified, the last scroll value entered is used; if a scroll value was never entered, it defaults to half a screen's worth of lines. Separate values are maintained for scrolling up and for scrolling down.

The scrolling and paging commands necessarily move the cursor within the text, but they cannot be used to home the cursor to an exact position at which changes are to be made. For this, you will have to use commands described in subsequent sections.

SEARCHING FOR STRINGS

As stated, Z uses the / string search command to scan forward looking for a string. This section discusses additional searching capabilities that Z provides, the capability of Z to match patterns called **REGULAR EXPRESSIONS**, and special characters that are used to match a class of characters.

Additional String-Searching Commands

The other string-searching commands are:

- | | |
|-----------------|---|
| ? | Similar to /, but Z searches backwards, rather than forward, to find the previous occurrence of the string. |
| n | Repeats the last string-search command. |
| N | Repeats the last string-search command, but in the opposite direction. |
| :se ws=0 | Turns the wrap scan option off. |
| :se ws=1 | Turns the wrap scan option on. |

When Z reaches the end or beginning of text without finding the string of interest, it normally wraps around to the opposite end of the text and continues the search. It does this because by default the wrap scan option is on. This option can be disabled by entering the set option command:

:se ws=0

thus causing the search to end when it reaches the end of text. The option can be reenabled by entering:

:se ws=1

Note that for this colon command, as for all colon commands, carriage return must be typed before the command is executed.

Regular Expressions

The strings you tell **Z** to search for are actually regular expressions, similar to the expressions or "patterns" that are used by the **grep** utility when matching strings. A regular expression is a pattern that is matched to character strings. The pattern can define a specific sequence of characters that make up the string; in this case, only that specific string matches the pattern. The pattern can also contain special characters that match a class of characters; in this case, the pattern can match any of a number of character strings.

For example, one such special construct is square brackets surrounding a character string; this matches any character in the enclosed string. So the regular expression

```
ab[xyz]cd
```

matches the strings

```
abxcd  
abycd  
abzcd
```

Another special character is *****, which matches any number of occurrences of the preceding pattern. For example, the regular expression

```
ab*c
```

matches many strings, including

```
abc  
abbc  
abxyzc
```

and so on. And the pattern

```
ab[xyz]*cd
```

matches many strings, including:

```
abcd  
abxcd  
abxycd  
abzzxcd
```

and so on.

The complete list of special characters and constructs that can be included in regular expressions is:

- `^` Matches the beginning of the line when it is the first character of a pattern.
- `$` Matches the end of the line when it is the last character of a pattern.
- `.` Matches any single character.
- `<` Matches the beginning of a word.
- `>` Matches the end of a word.
- `[str]` Matches any single character in the enclosed string.
- `[^str]` Matches any single character *not* in the enclosed string.
- `[x-y]` Matches any character between x and y.
- `*` Matches any number of occurrences of the preceding pattern.

Enabling Extended Pattern Matching

With **Z**, you can toggle the extent of pattern matching which will be done by **Z**. To have full pattern matching, type

```
:se ma=1
```

If you do not want full pattern matching, type

```
:se ma=0
```

which will only allow you to use `^` and `$` in regular expressions.

By default, extended pattern matching is disabled.

LOCAL MOVES

This section presents more commands for moving the cursor fairly short distances: up or down a few lines, along the line on which it is located, and so on. Some commands to accomplish this that we have already discussed are the CR (carriage return), space, and backspace. The commands introduced here reflect the importance of finely tuned, quickly executed movements.

Moving Around on the Screen:

Here are some commands for moving the cursor short distances:

- h** Moves to the left one character.
- j** Moves down one line, leaving the cursor in the same column.
- k** Moves up one line, leaving the cursor in the same column.
- l** Moves right one character.

The keys **^H**, **^J**, **^K**, and **^L** are synonyms for **h**, **j**, **k**, and **l**, respectively.

These commands can be preceded by a number that specifies the number of times the command is to be repeated.

Z has commands for moving the cursor to the top, middle, and bottom of the screen; they are **H**, **M**, and **L**, respectively. The cursor is positioned at the beginning of the line to which it is moved.

Remember the **-** command, which moved the cursor up a line, to the first nonwhitespace character? As you might expect, **+** moves the cursor down a line, to the first nonwhitespace character. **+** is thus equivalent to **CR**

Moving within a Line

The following commands have been discussed previously:

- | | |
|---|----------------------|
| h , ^H , backspace | Left one character. |
| l , ^L , space | Right one character. |

The following are a few more commands that allow you to move around on the current line:

- ^** Moves the cursor to the first nonwhitespace character on the line.
- 0** Moves the cursor to the first character on the line.
- \$** Moves the cursor to the last character on the line.

A few commands fetch another character from the keyboard, search for that character, beginning at the current cursor location, and leave the cursor near the character:

- f** Scan forward, looking for the character, and leave the cursor on it.
- t** Same as **f**, but leave the cursor on the character preceding the found character.
- F** Same as **f**, but scan backwards.
- T** Same as **t**, but scan backwards.
- ;** Repeat the last **f**, **t**, **F**, or **T** command.
- ,** Repeat the last **f**, **t**, **F**, or **T** command in the opposite direction.

Finally, the command **|** moves the cursor to the column whose number precedes the command. For example, the following command moves the cursor to column **56** on the current line:

56|

Word Movements

Z has several commands for moving the cursor to the beginning or end of a word that is near the cursor:

- w** Moves to the beginning of the next word (alphanumeric only).

- b** Moves to the beginning of the previous word (alphanumeric only).
- e** Moves to the end of the current word (alphanumeric only).

For the preceding commands, a word is defined in the normal way: a string of alphabetical and numerical characters surrounded by whitespace or punctuation. There is a variant of each of these commands, differing only in the definition of a word: They think that a word is any string of nonwhitespace characters surrounded by whitespace. The variant of each of these commands is identified by the same letter, but in uppercase instead of lowercase:

- W** Moves to the beginning of the next word (any characters surrounded by whitespace).
- B** Moves to the beginning of the previous word (any characters surrounded by whitespace).
- E** To the end of the current word (any characters surrounded by whitespace).

Each of these commands can be preceded by a number, specifying the number of times the command is to be repeated. For example,

- 5w** Moves forward five words.

The word movement commands cross line boundaries, if necessary, to find the word they are looking for.

Moves within C Programs

Z has several commands for moving the cursor within C programs:

-]] and [[** Moves to the opening curly brace, {, of the next or previous function, respectively.
- %** Moves to the parenthesis, square bracket, or curly bracket that matches the one on which the cursor is currently located.

{ and } Moves to the preceding or next blank line.

The **[** and **]** commands assume that the opening and closing curly braces for a function are in the first column of a line, and that all other curly braces are indented.

As an example of the **%** command, given the statement:

```
while ((a = getchar()) != EOF) && (c != 'a')
```

with the cursor on the parenthesis immediately following the **while**, the **%** command will move the cursor to the last closing parenthesis on the line.

Marking and Returning

Z has commands that allow you to set markers in the text and later return to a marker. Twenty-six markers are available, identified by the alphabetical letters.

Unlike the other commands described in this section, these commands are not limited to moves within the current area of the cursor—they can move the cursor anywhere within the text.

A marker is set at the current cursor location using the command

mx

where *x* is the letter with which you want to mark the location.

There are two commands for returning to a marked position:

\x Moves the cursor to the location marked with the letter *x*

'x Moves the cursor to the first nonwhitespace character on the line containing the *x* marker.

Occasionally, you may accidentally move the cursor far from the desired position. There are two single quote commands for returning you to the area from which you moved:

" Returns the cursor to its exact starting point.

" Returns the cursor to the first nonwhitespace character on the line from which the cursor was moved.

For example, if the cursor is on the line:

```
if (a >= 'm' && a <= 'z')
```

at the character <, then following a command which moves the cursor far away, the command `` will return the cursor to the < character, and the command '' will return it to the beginning of the word `if`.

Adjusting the Screen

The **Z** command is used to redraw the screen, with a certain line at the top, middle, or bottom of the screen.

To use it, place the cursor on the desired line, then enter the **Z** command, followed by one of these characters:

- CR** To place the line at the top of the screen.
- To place it at the bottom.
- .** To place it in the middle of the screen.

The **Z** command is not a true cursor motion command, because the cursor is in the same position in the text after the command as before.

Control- L repaints the screen.

MAKING CHANGES

The previous section described the cursor movement commands. The next several sections describe commands for making changes to the text.

Small Changes

This section describes several more small commands. The first two commands listed below were already discussed in a previous section.

- x** Which deletes the character at which the cursor is located.
- dd** Which deletes the line at which the cursor is located.

- X** Delete the character which precedes the cursor. Can be preceded by a count of the number of characters to be deleted.
- D** Delete the rest of the line, starting at the cursor position.
- rx** Replace the character at the cursor with **x** .
- R** Start overlaying characters, beginning at the cursor. Type the escape key to terminate the command.
- s** Delete the character at the cursor and enter insert mode. When preceded by a number, that number of characters is deleted before entering insert mode.
- S** Delete the line at the cursor and enter insert mode; when preceded by a count, that number of lines is deleted before entering insert mode.
- C** Delete the rest of the line, beginning at the cursor, and enter insert mode.
- J** Join the line on which the cursor is positioned with the following line.

Operators for Deleting and Changing Text

Z has a small number of commands for modifying text. They all have the same form, consisting of a single letter command, optionally preceded by a count and always followed by a cursor motion command. The count specifies the number of times the command is to be executed. The command affects the text from the current cursor position to the destination of the cursor motion command, if the starting and ending position of the cursor are on the same line. If these positions are on different lines, the command affects all lines between and including the lines which contain the starting position and ending positions.

In this section, we are going to describe the operators for deleting and changing text, **d** and **c**:

- d** Deletes text as defined by the cursor motion command.

- c** Same as **d**, but **Z** enters insert mode following the deletion.

For example,

- dw** Deletes text from the current cursor location to the beginning of the next word.
- 3dw** Deletes text from the cursor to the beginning of the third word.
- d3w** Same as **3dw**.
- db** Deletes text from the current to the beginning of the previous word.
- d'a** Deletes text from the cursor to the marker **a**, if the marker and the starting cursor position are on the same line. Otherwise, deletes lines from that on which the cursor is located through that on which the marker is located.
- d/var** Deletes text either from the cursor to the string **var** or between the lines at which the cursor is currently located and that on which the string is located.
- d\$** Deletes the rest of the characters on the line, and hence is equivalent to **D**.

Deleting and Changing Lines

Previously, we presented a command for deleting lines: **dd**. As you can see now, this is a special form of the **d** command, because the character following the first **d** is not a cursor motion command.

For all the operator commands, typing the command character twice will affect whole lines. Thus, typing **cc** will clear the line on which the cursor is located and enter insert mode. Preceding **cc** with a number will compress the specified number of lines to a single blank line and enter insert mode on that line.

Moving Blocks of Text

When text is deleted using the **d** or **c** command, it is moved to a buffer called the unnamed buffer. (There are other buffers available, which have names. More about them later).

Data in the unnamed buffer can be copied into the main text buffer using one of the **put** commands:

- p** Copies the unnamed buffer into the main text buffer, after the cursor.
- P** Same as **p**, but the text is placed before the cursor.

Thus, the delete and put commands together provide a convenient way to move blocks of text within a file.

The contents of the unnamed buffer is very volatile: When any command is issued that modifies the text, the text which was modified is placed in the unnamed buffer. This is done so that the modification can be undone, if necessary, using one of the undo commands. For example, if you delete a character using the **x** command, the deleted character is placed in the unnamed buffer, replacing whatever was in there. The unnamed buffer only holds the contents of the last command executed. So you have to be careful when moving text via the unnamed buffer. If you delete text into the unnamed buffer, expecting to place it elsewhere, then issue another command which modifies the unnamed buffer before issuing the **put** command. The deleted text is no longer in the unnamed buffer.

As you will see, the named buffers can also be used to move blocks of text, and their contents are not as volatile.

Duplicating Blocks of Text: the Yank Operator

The yank operator, **y**, copies text into the unnamed buffer without first deleting it from the main text buffer. When used with the **put** command, it provides a convenient way for duplicating a block of text.

The **y** operator has the same form as the other operators: an optional count, followed by the **y** command, followed by a cursor motion command. The command yanks the text from the cursor position to the destination of the cursor motion command, if the starting and ending positions are on the same line. If they are on separate lines, a whole number of lines are yanked,

from the cursor position through the point the cursor would be moved to by the cursor motion command. The text is yanked into the unnamed buffer.

For example,

yw Copies text from the cursor to the next word into the unnamed buffer.

y3w Copies text from the cursor to the beginning of the third word.

3yw Same as **y3w**.

y'a Copies text from the cursor location to the marker **a** into the unnamed buffer, if the two positions are on the same line. Otherwise, copies entire lines between and including those containing the two positions.

As a special case, the command **yy** will yank a specified number of whole lines. The command **Y** is a synonym for **yy**. For example,

yy Yanks the line at the current cursor location.

y3w Yanks three lines, beginning with the one at the cursor location.

Named Buffers

In addition to the unnamed buffers, **Z** has 26 named buffers, each identified by a letter of the alphabet, which can be used for rearranging text. Text can be deleted or yanked into a named buffer and put from it back into the main text buffer.

The advantage of these buffers over the unnamed buffer in rearranging text is that their contents are not volatile: When you put something in a named buffer, it stays there and will not be overwritten unexpectedly. Also, as you will see, the named buffers can be used to move text from one file to another.

To yank text into a named buffer, use the yank operator, preceded by a double quote and the buffer name, and followed by a cursor motion

command. For example, the following will yank three words into the **a** buffer:

"ay3w

and the following yanks four lines into the **b** buffer, beginning with the line on which the cursor is located:

"b4yy

Text is deleted into a named buffer in the same way: The delete command is used, preceded by a double quote and the buffer name. For example, to delete characters from the cursor to the **a** marker into the **h** buffer:

"hd 'a

The preceding command, when the source and destination cursor positions are on separate lines, will delete a number of whole lines into the **h** buffer, from that on which the cursor is initially located through that containing the destination position.

To delete ten lines into the **c** buffer:

"c10dd

Text in a named buffer is put back into the main text using the **put** commands **p** and **P**, preceded by a double quote and the buffer name. For example:

"ap puts text from the **a** buffer, after the cursor.

"zP puts text from the **z** buffer, before the cursor.

Moving Text between Files

The named buffers are conveniently used to move text from one file to another. First yank or delete text from one file into a named buffer; then switch and begin editing the target file, using the **:e** command:

:e filename

(More on this later). Then move the cursor to the desired position and put text from the named buffer.

Shifting Text

The shift operators, < and > , which are used to shift text left and right a tab stop, respectively.

For example,

>/str

shifts right one tab stop the lines from that on which the cursor is located through that containing the string **str**.

Following the standard operator syntax, repeating the shift operator twice affects a number of whole lines:

5<< Shifts five lines left.

>> Shifts one line right.

Undoing and Redoing Changes

Z remembers the last change you made, and has a command, **u**, which undoes it, restoring the text to its original state.

Z also remembers all the changes which were made to the last line which was modified. Another **UNDO** command, **U**, undoes all changes made to that line.

Finally, the period command, **.**, reexecutes the last command that modified text.

INSERTING TEXT

The following commands have been discussed:

- a** Append after cursor.
- i** Insert before cursor.
- o** Open new line below cursor.
- O** Open new line above cursor.
- C** Delete to end of line, then enter insert mode.

s Delete characters, then enter insert mode.

S Delete lines, then enter insert mode.

This section discusses the remaining commands for entering insert mode and describes some other features of insert mode.

Additional Insert Commands

The other commands for entering insert mode are:

A Append characters at the end of the line on which the cursor is located. This is equivalent to **\$a**.

I Insert before the first nonwhitespace character on the current line. This is equivalent to **^i**.

Insert Mode Commands

Some editing can be done on text entered during insert mode, using the following control characters:

backspace Delete the last character entered.

^H Same as "backspace" character.

^D Same as "backspace" character.

^X Erase to beginning of insert on current line.

^V Enter next character into text without attempting to interpret it.

^V is used to enter nonprinting characters into the text. For example, to enter the character Control-A into the text, type

^V^A

That is, hold down the control key, then type the **V** key, then the **A** key, then release the control key. As mentioned earlier, nonprinting characters are displayed as two characters: "**^**" followed by a character whose ASCII code equals that of the nonprinting character plus 0x40.

Autoindent

The **Z** autoindent option is useful when entering C programs. When you are in insert mode and type the carriage return key with the autoindent option enabled, the cursor automatically indents on the new line to the same column on which the first nonwhitespace character appeared on the previous line. This feature is useful for editing C programs because it encourages statements that are part of the same compound statement to be indented the same amount, thus making the program more readable.

Z autoindents a line by inserting tab and space characters at the beginning of a new line. If you do not want the lines indented that much, backspace over these automatically inserted tabs and spaces until you reach the desired degree of indentation.

The autoindent option can be selectively enabled and disabled using the set options command:

:se ai=0 Disables autoindent.

:se ai=1 Enables autoindent.

When **Z** is activated, autoindent is enabled.

MACROS

Z allows you to define a sequence of commands, called a **MACRO**, and then execute the macro one or more times.

When a macro is defined to **Z**, it is placed in a special buffer, called the macro buffer, and then executed once. There are two ways to define a macro to **Z**: immediately and indirectly.

Immediate Macro Definition

An immediate macro definition is initiated by typing the characters

:>

Z responds by clearing the status line, displaying these characters on the line, and waiting for you to enter the sequence of commands.

As you enter the commands, **Z** displays them on the status line and enters them immediately into the macro buffer, hence the term IMMEDIATE MACRO DEFINITION.

If you make a mistake while entering commands, you can simply backspace and enter the correct characters.

To terminate the definition, type the carriage return key. **Z** then executes the sequence of commands in the macro buffer. The contents of this buffer are not altered by executing the macro, so you can reexecute the macro without reentering it, as described below.

Examples

The following macro advances the cursor one line and deletes the first word on the new line:

```
:>+dw
```

This macro contains two commands: **+**, which advances the cursor, and **dw**, which deletes the word beneath the cursor.

The next macro moves the cursor to the previous line and deletes the last character on the line:

```
:>-$x
```

It contains three commands: **-**, which moves the cursor to the previous line; **\$**, which moves the cursor to the last character on that line; and **x**, which deletes the character beneath the cursor.

You can also insert text using a macro. You enter insert mode using one of the normal insert commands. The characters that follow the insert command on the macro line, up to a terminating escape character, are then inserted into the text. The escape character causes **Z** to return to command mode and continue executing commands in the macro that follow the insert command.

For example, the following macro advances the cursor to the next line, deletes the second word on the line, inserts the character string "and furthermore", and deletes the last word on the line:

```
:>+wdwiand furthermore <ESC>$bdw
```

The last macro contains the following commands:

- +** Advances the cursor to the next line.
- w** Moves the cursor to the second word on the line.
- iand furthermore <esc>**
Inserts the text **and furthermore** . **<ESC>** stands for the escape key.
- \$** Moves the cursor to the last character on the line.
- b** Moves the cursor to the beginning of the last word on the line.
- dw** Deletes the word beneath the cursor.

Z also allows you to search for a string from within a macro. Enter the string search command in the macro (for example, **/**), followed by the string, followed by the ESC character. For example, the following macro moves the cursor to the word **Melinda** and deletes it:

```
:>/Melinda<ESC>dw
```

It contains the commands

- /Melinda<ESC>** Moves the cursor to **Melinda** . **<ESC>** stands for the escape key.
- dw** Deletes **Melinda**.

The following macro finds **Melinda** and replaces it with **John** :

```
:>/Melinda<ESC>cwJohn<ESC>
```

It contains the commands:

- /Melinda <ESC>** Moves the cursor to **Melinda**
- cwJohn <ESC>** Changes **Melinda** to **John**.

Indirect Macro Definition

The other way of defining a macro is to yank a line containing a sequence of commands from the main text buffer into a named buffer.

Commands for indirect macro definition are:

@x Causes **Z** to move the contents of the **x** buffer to the macro buffer and then execute it once.

"xv A synonym for **@x**.

Indirect macro definition of macros has several advantages over immediate definition: For one, if a macro defined immediately is incorrect, you have to reenter the entire macro. With an indirectly defined macro, you can edit the macro definition in the main text buffer and then move it back to the macro buffer.

Another advantage is that you can store several macros in the named buffers and easily reexecute a macro, without having to reenter it. With immediate definition, when a new macro is defined, the previously defined macro is lost and must be reentered to be reexecuted.

One difference between entering macros immediately and via the named buffer concerns the method for specifying the end of a search string and for exiting insert mode. With immediate definition, you do this by typing the ESC key directly. For indirect definition, in which the macro is first entered into the main text buffer, typing the ESC key would cause **Z** to exit insert mode, not to enter the ESC key into the text of the macro. In this case, you enter the ESC key by first typing Control-V, then ESC. This causes **Z** to enter the ESC character into the text of the macro and remain in insert mode.

Reexecuting Macros

Once a macro is defined and is in the macro buffer, it can be reexecuted by typing one of the commands:

@
v

Preceding the command with a count causes the macro to be executed the specified number of times.

Wrapping Around During Macro Execution

While executing a macro, **Z** may reach the beginning or end of the text, and want to continue beyond that point. This is especially true when reexecuting macros. The macro wrap option, **wm**, specifies whether **Z** should terminate the macro execution at that point, or continue at the opposite end of the text.

This option is enabled and disabled using the set options command:

:se wm=0 Disables macro wrapping.

:se wm=1 Enables macro wrapping.

When **Z** starts, this option is enabled.

EX-LIKE COMMANDS

The **SUBSTITUTE** and **REPEAT LAST SUBSTITUTE** commands are a set of commands in the **Z** editor that are similar to commands in the UNIX **Ex** editor. This section describes the syntax for these commands, and then gives details about the substitute and repeat last substitute commands.

The ex-like commands consist of a leading colon, followed by zero, one, or two addresses identifying the lines to be affected by the command, followed by a single-letter command, followed by command parameters, and terminated by a carriage return. Most commands have a default set of lines that they affect, thus frequently allowing you to enter commands without explicitly specifying a range.

These commands support regular expressions, as defined in the **Z** documentation, for identifying addresses and strings to be searched for.

Addresses in Ex Commands

An address can be one of the following:

- A period, `.`, addresses the current line; that is, the line on which the cursor is located.
- The character `$` addresses the last line in the edit buffer.
- A decimal number n addresses the n -th line in the edit buffer.
- `'x` addresses the line marked with the mark name `x`. Lines are marked with the `m` command.
- A regular expression surrounded by slashes (`/`) addresses the first line containing a string that matches the regular expression. The search begins with the line following the current line and continues toward the end of the edit buffer. If a line is not found when the end of the buffer is reached, and if the **Z** option `ws` is set to 1 (i.e., by the `:se ws=1` command), the search continues at the beginning of the buffer, stopping when the current line is reached.
- A regular expression surrounded by question marks (`?`) also addresses the first line containing a string that matches the regular expression. But in this case, the search begins with the line preceding the current line in the edit buffer and continues towards the beginning of the buffer. If a line is not found when the beginning of the buffer is reached, and if the **Z** option `ws` is set to 1 (i.e., by the `:se ws=1` command), the search continues at the end of the buffer, stopping when the current line is reached.
- An address followed by a plus or a minus sign, which in turn is followed by a decimal number n addresses the n -th line following or preceding the line identified by the address.

When two addresses are entered to define the range of lines affected by a command, the addresses are usually separated by a comma. They can also be separated by a semicolon; in this latter case, the current line is set to the line defined by the first address, and then the line corresponding to the second address is located.

When no value is specified for the first address in an address range, it is assumed to be the current line or the first line in the buffer, depending on whether the second address was preceded with a comma or a semicolon. When no value is specified for the second address in an address range, it is assumed to be the last line in the buffer. Thus, if neither the beginning nor the ending address of a range is specified, the range consists of either all the lines in the buffer or the lines from the current through the last line in the buffer, depending on whether comma or semicolon is used to separate the unspecified addresses.

The Substitute Command

The substitute command has the following form:

`:[range]s /pat /rep / [options]`

where square brackets surround a parameter to indicate that the parameter is optional.

Z searches the lines specified by *range* for strings that match the regular expression *pat*, replacing them with the *rep* string. If *range* is not specified, only the current line is searched. When the command is completed, the cursor is left on the character following the last replaced string.

The c Option

Normally, **Z** automatically replaces a string that matches *pat*. Specifying **c** as an option causes **Z** instead to pause when it finds a matching string, ask if you want the string to be replaced, and make the replacement only if you give your permission.

The g Option

Z replaces only the first *pat*-matching string on a line. Specifying **g** as an option causes **Z** instead to replace all matching strings on a line. In this case, after **Z** replaces a string on a line, it continues searching for more strings on the line at the character following the replaced string.

An ampersand (&) in the replacement string *rep* is replaced by the string that matched *pat*. The special meaning of & can be suppressed by preceding it with a backslash, \.

A replacement string consisting of just the percent character (%) is replaced in the current substitution by the replacement string that was used in the last substitution. The special meaning of % can be suppressed by preceding it with a backslash, \.

Examples

:s/aBD/def/

Search the line on which the cursor is located for the string **aBD**; if found, replace it with the string **def**.

:1,\$s/ab*c/xyz/

Search all lines in the edit buffer for strings that begin with **a**, end in **c**, and have zero or more **b**'s in between; replace such strings with **xyz**. On any given line, only the first occurrence of a string that matches the pattern is replaced.

:/{/;/}/s/for/while/c

Find the first line following the current line that contains a { ; then find the first line following this line that contains a } . In the lines between and including these lines, search for the string **for** . For each such string, ask if it should be replaced; if yes, replace it with **while** .

The "&" (Repeat Last Substitute) Command

The **&** command has the form

:[range]&

where brackets indicate that the parameters are optional.

The **&** command causes the last substitute command to be executed again, using the same search pattern, replacement string, and options as were used in the previous command. The command searches the lines that are specified in the **&** command range; if *range* is not specified, the substitution is performed on only the current line.

QUIKFIX COMMANDS

With version 5 of Aztec C, **z** now supports the following five commands to be used in conjunction with the **QuikFix** facility.

- :cc** redisplay current error
- :cp** display previous error
- :cn** display next error

Using any of the above commands will load the source file if not already loaded and move to the appropriate line and column and display the error message.

- :cf** This command is used to parse the next set of errors from the **AztecC.Err** file. Saying **z -e** is the same as saying **z** and then giving the **:cf** command. This can be used if more than 25 errors are in the error file or if the error file contains errors for more than one source file.
- :cq** This command works the same as the **:q** command to exit **z**, except that it causes **z** to return an error code instead of the normal non-error code. This is used to cause the compiler not to recompile the module when **z** terminates

For additional information on the **QuikFix** facility, refer to the **Getting Started** Chapter of the *Aztec C User Manual*

THE OPTION FILE

Z contains several options for controlling its operation in different situations, including the autoindent and macro wrap. (This manual contains a complete list of these commands at the end of this chapter.) This section presents another feature of **Z** related to options; i.e., the ability to set options automatically, when **Z** is started.

When **Z** starts, it reads options from the file specified by the **ZOPT** environment variable, if it exists.

The environment variable **ZOPT** defines the name of the options file.

Each line in the options file defines the value of one option, with a statement of the form

opt = val

where *opt* is the name of the option, and *val* is its value. For example, the following sets the tab-width option to eight characters:

ts=8

The ZOPT Environment Variable

The ZOPT environment variable defines the file in which **:set** options are found.

For example, the following command says that the options are in the file **zopt.cmd**:

set ZOPT=zopt.cmd

Setting Options for a File

When **Z** makes a file the edit file by reading it into the edit buffer, the file itself specifies the options to be in effect during its edit session. This feature is most useful in editing files that have different tab settings.

A file specifies option values by including strings of the form

:opt=val

in the first ten lines of the file. For example, the following line could be used near the front of a C program, causing a tab width of eight characters to be used:

/* :ts=8 */

When **Z** starts editing a file, the tab width is set back to the default value, four characters, before the file is scanned for option settings.

STARTING AND STOPPING Z

You already know how to start and stop **Z**. This section presents more information related to starting and stopping **Z**.

Starting Z

Previously, we said that **Z** was started by specifying the name of the file to be edited on the command line:

Z *filename*

You may also start **Z** without specifying a *filename* or by specifying a list of files to be edited.

Starting Z without a Filename

Z can be started without specifying a filename, by entering the command:

Z

When you start **Z** without specifying *filename*, once it is active you normally tell **Z** the name of the file to be edited using the **:e** command:

:e *filename*

However, it is not necessary for **Z** to know the name of the file you are editing immediately upon opening; **Z** allows you to create and modify text in the text buffer without knowing the name of the file to which you intend to write the text. But you must explicitly tell **Z** to write the text to *filename* when you want to save the text that you worked on. You would use the command

:w *filename*

Z cannot automatically write the text, because it does not know which file you are editing.

Starting Z with a List of Files

You may start **Z** and pass it a list of names of files to be edited, as follows:

Z *file1 file2 ...*

Z remembers the list, and makes the first file in the list the edit file; that is, reads the file into the main text buffer and allows it to be edited.

Z contains a command, **:n**, that makes the next file in the list the edit file, after writing the contents of the text buffer back to the current edit file. File lists are discussed in more detail below.

Stopping Z

Previously, we presented the following commands for stopping Z:

- zz** If the file text in the edit buffer is modified, Z writes the text to the file, after changing the extension of the original file to **.bak**.
- :q!** Stops Z without writing the text to the file.

Three other commands for exiting Z are:

- :wq** Writes the text to the buffer, similar to, except that the text in the main text buffer is always written to the file, even if no changes have been made.
- :q** Conditionally stops Z. If no changes were made to the file text, Z stops; otherwise, it displays a message and remains active.
- :cq** Works the same as the **:q** command to exit the editor, except that it causes the editor to return an error code instead of the normal non-error code. This is used to tell the compiler not to recompile the module when the editor terminates.

ACCESSING FILES

This section discusses some additional commands for accessing files. For example, Z usually knows the name of the file you are editing, and in the sections that follow we will call this the edit file. Z makes use of this knowledge, allowing you to write to the edit file without specifying it by name. For example, the **ZZ** command writes text to the edit file without requiring you to enter the name of the file.

Some commands allow you to access files without redefining Z's idea of the edit file. The commands described in the next two subsections fall into this category.

Other commands cause Z to terminate editing of one file and begin editing another; this new file becomes the edit file. The commands described in the other subsections of this section are of this type.

Filename

In the **Z** commands that require a filename, you enter the name using the standard system conventions. However, some characters are special to **Z**:

- # Refers to the last edit file.
- % Refers to the current edit file.
- \ Causes the next character to be used in the filename and not be interpreted.

To enter a filename that contains these characters, precede the special character with the character "\".

Writing Files

The command **:w** writes the contents of the main text buffer to a file, without redefining the identity of the current edit file. It has the following forms:

- :w** Write to the current edit file.
- :w filename** Write to the specified file
- :w filename** Same as **:w filename**, but the file is overwritten if it exists.

As with all colon commands, carriage return must be typed to cause **Z** to execute the command.

When entered without a filename, **:w** creates a new file having the name of the current edit file and writes the contents of the edit buffer to it. This form of the **:w** command is commonly used to periodically save text during a long edit session, as protection against possible system failures.

The option **bk** tells **Z** whether it should save the original edit file before creating a new one. If **bk** is 1, the original is saved, and if 0, it is not. **Z** saves the original file by changing its name to **.bak**. An existing **.bak** file is erased before the rename occurs.

When a filename is entered with the **:w** command, the text is written to that file if it does not already exist. If it does, nothing is written and **Z** displays a message on the status line; in this case you must use the **:w!** form of the command to overwrite the file.

The **:w!** command unconditionally writes the text to the specified file after truncating the file, if it exists, so that nothing is in it. Unlike the **:w** command that does not specify a filename, the **:w!** command does not save the original file as a **.bak** file.

Reading Files

The command

:r filename

merges one file with a file being edited, without redefining the identity of the edit file. It reads the contents of the specified file into the main text buffer, inserting the new text following the line on which the cursor is located. It does not alter text that is already in the edit buffer.

Editing Another File

The following commands cause **Z** to stop editing one file and begin editing another, which then becomes the edit file:

- | | |
|---------------------|---|
| :e filename | Edit the specified file. |
| :e! filename | Edit the file, discarding changes to the current edit file. |
| :e | Reload the current edit file. |
| :e! | Reload the current edit file, discarding changes. |
| :e # | Edit the previous edit file. |
| ^^ | Synonym for :e # . (the command is control-^). |

Z begins editing another file by erasing the contents of the main text buffer, resetting the tab width to four characters, redrawing the display with the first screenful of lines from the file, and setting the cursor at the first character in the text.

When switching to a new edit file, **Z** does not change the contents of the named and unnamed buffers. Thus, these buffers can be used to hold text that is to be moved from one file to another and to contain commonly used macros.

The command

:e filename
causes the specified file to conditionally become the edit file. The condition is that changes must not have been made to the text of the current edit file since it was last written to disk. If this condition is met, then the switch is made; otherwise, **Z** displays a message on the status line and nothing is changed: The identity of the edit file is the same, the contents of the edit buffer are not modified, and the options are not changed.

If **Z** does not let you switch edit files when you enter

:e filename

and you want to save the changes to the current edit file, enter the sequence:

:w

:e filename

You can unconditionally cause **Z** to begin editing a new file by entering:

:e! filename

In this case, **Z** does not care whether or not you made changes to the current edit file since it was last written to disk; it begins editing the new file without changing the previous edit file.

Sometimes the text in the edit file may get hopelessly scrambled, and you want to get a fresh copy of the edit file contents. The command

:e!

specified without a filename does just that.

Z not only remembers the name of the current edit file you are editing; it remembers the name of the last file you edited as well. **Z** allows you to refer to this name using the character **#** in **:e** commands, thus providing a quick means to reedit the previous edit file:

:e #

causes the previous edit file to conditionally become the current edit file, and

:e! #

causes it to unconditionally become the edit file.

The command ^^ (that is, Control-^) is a synonym for :e #.

Z also remembers the position at which the cursor was located in the previous edit file, and when you begin reediting this file it sets the cursor back to this position.

File Lists

Z's file list feature is convenient to use when you have several files to edit. You pass **Z** a list of the files and begin editing the first one. When you are finished with one file, a command switches to the next file in the list, after you have explicitly saved the changes to the current edit file. An option to the command prevents **Z** from saving changes, and another command rewinds the file list so that you are back editing the first file in the list again.

There are two ways to pass the list of files to be edited to **Z**: as parameters to the command that starts **Z**, and as parameters to the :n command. In each case, **Z** remembers the list and makes the first file in the list the edit file. For example,

```
z file1 file2 file3
```

starts **Z** and defines the list of files—**file1**, **file2**, and **file3**. **Z** makes **file1** the edit file; that is, prepares it for editing by reading it into the edit buffer and displaying its first lines.

When **Z** is active, the command

```
:n file4 file5 file6
```

defines a new list of files—**file4**, **file5** and **file6**. **Z** makes **file4** the edit file.

When used without a files list, the :n command switches from one file in the list to the next:

```
:n!
```

Switches without writing anything to the current edit file.

The :rew command rewinds the file list, i.e., makes the first file in the list the edit file. This command behaves like the :n command, in that any change to the current edit file must be rewritten before rewinding; and

when an exclamation mark is appended to the command, the rewind occurs, regardless of the state of the current edit file.

Tags

Z has a feature useful for editing large C programs that contain many functions distributed over several files. With the aid of a cross-reference file relating tags, (i.e., function names), to the files containing them, you simply tell **Z** the name of the function that you want to edit and **Z** makes the file containing it the edit file by reading it into the edit buffer and positioning the cursor to the function.

The following commands specify the tag of the function to be edited:

- | | |
|-----------------|---|
| :ta tag | Position to the function named tag in the appropriate file, if the current edit file is up to date |
| :ta! tag | Same as :ta tag , but the switch to the new file occurs even if the current edit file is not up to date. |

When using the **:ta** command, the current edit file is considered up to date if the text in the edit buffer has not been modified since it was last written to the file. When used without the trailing **!**, the **:ta** command does not switch edit files if the current edit file is not up to date; it only displays a message on the status line. You can then either write the text in the edit buffer to the file and reenter the **:ta** command, or immediately enter the **:ta!** command, to switch edit files anyway.

If *tag* ends up in the current file, it works regardless of the current file's modification status.

The command

^]

Control-], is convenient when, while editing or viewing one function, you want to edit or examine a function that it calls. You just set the cursor to the name of the called function and enter **^]**; **Z** makes the file containing the called function the edit file, and positions the cursor to this function.

For example, while examining the file **crtadvr.c**, you may come across a call to the function **pcadvr**, and may want to take a look at it. By positioning the cursor at the beginning of the word **pcadvr** and typing **^]**, **Z** makes

the file containing **pcdvx** the edit file and leaves the cursor positioned at this function.

The ctags Utility

The utility program **ctags** creates the cross reference file, **tags**, that relates function names to the file containing them. **ctags** is activated by a command of the form

ctags file1 file2 ...

where file1, ..., are names of files whose functions are to be placed in the cross reference file. A filename can specify a group of files using the character *. For example:

***.c**

specifies all files whose extension is **.c**, and

f*.c

specifies all files whose first character is **f** and whose extension is **.c**.

ctags creates the cross reference file, **tags**, in the current directory on the default drive.

When a tags command is given, **Z** searches for this file in the current directory.

OPTIONS

Z provides several options under user control that define how **Z** behaves in certain situations. Most of these options have been discussed peripherally in previous sections, when appropriate. This section focuses on the options.

Each option is identified by a code. The options and their codes are:

- ai** Auto-indent option. When this option is enabled and you begin inserting text on a new line, **Z** automatically indents the line by inserting tabs and spaces so that the text you type is correctly aligned with the text in the line above it. By default, this option is enabled.

- eb** Error bells option. When this option is enabled, **Z** beeps when you make a mistake. By default, this option is enabled.
- ma** Magic option. When this option is enabled, regular expressions used in string searches can include extended pattern matching characters. Otherwise, only the characters **^** and **\$** are special and the extended pattern matching constructs are gotten by preceding them with ****. By default, this option is disabled.
- ts** Tab Set Option. Specifies the number of characters between tab settings. By default, the tab width is four characters.
- wm** Wrap On Macro Option. When this option is enabled, and a macro being executed reaches the end of the buffer, the macro wraps around to the beginning of the buffer and continues. By default, this option is enabled.
- ws** Wrap On Search Option. When this option is enabled and a search for a string reaches the end of the buffer without finding the string, the search continues at the opposite end of the buffer. By default, this option is enabled.
- bk** Defines whether **Z**, when a **:w** command is entered to write the edit buffer to the current edit file, should save the original edit file before creating a new one.
- cs** Clear Screen Option. When this option is enabled, the repaint command (Control-L) will first clear the entire screen and then repaint it. If it is disabled, each line of text on the screen will be redrawn and then cleared to the end of the line. Some systems redraw the screen much faster if this option is enabled.
- sm** Silent Macro Option. When this option is enabled, macros operate silently. If it is disabled, macros display

their commands as they execute. The main advantage to silent operation is that it is faster.

An option is enabled by setting it to 1, and disabled by setting it to 0.

DIFFERENCES BETWEEN Z AND VI

Z is very similar to the UNIX editor **Vi**, in the following ways:

- Both are full-screen editors, display text in the same way, and reserve one line of the display for messages;
- They have the same two modes: command and insert;
- **Z** supports most of the **Vi** commands. The **Z** commands are activated by the same keystrokes and perform the same functions as their **Vi** counterparts.

Z and **Vi** differ in the following ways:

- In **Z**, the buffer in which text is edited is entirely within RAM memory; in **Vi**, the buffer is both in memory and on disk. Because of this, **Z** is restricted in the size of program that can be edited, but **Vi** is not;
- A single copy of **Vi** can be configured to use any type terminal. A single copy of **Z** is pre-configured to use just one terminal;
- **Vi** has an underlying editor, **ex**, whose commands can be executed while **Vi** is active. **Z** does not have an underlying editor. However, **Z** does support some **ex** commands directly; these are the commands whose first character is ":". (**Vi** interprets the ":" as a request to execute the **ex** command which is entered after the ":");
- **Vi** has commands and options useful for editing documents and for editing LISP programs, but **Z** does not;
- With **Vi**, you can create a shell and suspend **Vi** while executing commands from within the new shell. With some **Vis**, you can

also suspend Vi while executing commands from the shell that activated Vi. Z does not support either of these features;

- Vi saves the last nine deleted blocks of text and has commands with which it can recover them, if necessary. Z lets you recover the last deleted block;
- With Vi, operator commands can affect exactly the characters between the starting and ending cursor positions, even when the positions are on different lines. Z has variations of these commands which allow whole lines to be affected, between and including the lines containing the two positions.

In Z, operator commands in which the starting and ending cursor positions are on different lines always affect whole lines, between and including the lines containing the two positions.

Starting Z

The Display

Options

Adjusting the Screen

^F forward screenful
^B backward screenful

^D	scroll down half screen
^U	scroll up half screen
zCR	redraw, current line at top
z-	redraw, current line at bottom
z.	redraw, current line at center

Positioning within File

g	go to line (default is end of file)
G	go to line (default is beginning of file)
/pat	move cursor to <i>pat</i> searching forwards
?pat	move cursor to <i>pat</i> searching backwards
n	repeat last / or ?
N	repeat last / or ? in reverse direction
]]	next "^{"
[[	previous "^{"
%	find matching (), {}, or [].

Marking and Returning

“	previous context
”	first nonwhite at previous context
mx	mark position with letter <i>x</i>
'x	to mark <i>x</i>
'x	first nonwhite at mark <i>x</i>

Line Positioning

H	top of screen
M	middle of screen
L	bottom of screen
+	next line, first nonwhite
CR	next line, first nonwhite
-	previous line, first non-white
LF	next line, same column
j	next line, same column
^K	previous line, same column

k previous line, same column

Character Positioning

0 beginning of line
^ first nonwhite at beginning of line
\$ end of line
space forward a character
^L forward a character
l forward a character
^H backwards a character
h backwards a character
fx find character *x* forward
Fx find character *x* backwards
tx position before character *x* forward
Tx position before character *x* backwards
; repeat last f, F, t or T
, repeat last f, F, t or T in reverse direction
| move to specified column number

Words and Paragraphs

w word forward
W blank delimited word forward
b back word
B back blank delimited word
e end of word
E end of blank delimited word
} to next blank line
{ to previous blank line

Insert and Replace

a append after cursor
A append at end of line
i insert before cursor
I insert before first non-blank in line

o	open line below current line
O	open line above current line
rx	replace single character with x
R	replace characters

Corrections During Insert

^H	erase last character
^D	erase last character
^X	erase to beginning of insert on current line
^V	insert following character directly
^W	delete previous word typed

Operators

d	delete
c	delete and insert
<<	left shift
>>	right shift
y	yank

Miscellaneous Operations

D	delete rest of line
C	change rest of line
s	substitute characters
S	substitute lines
J	join lines
x	delete characters starting at cursor
X	delete characters before cursor
Y	yank lines

Yank and Put

p	put after current
P	put before current
"xp	put from buffer x
"xy	yank to buffer x
"xd	delete to buffer x

Undo and Redo

u	undo last change
U	restore current line
.	repeat last change command

Macros

@X	execute macro in buffer x
"xv	execute macro in buffer x
@@	repeat last macro
v	repeat last macro

QuikFix Commands

:cc	redisplay current error
:cn	display next error
:cp	display previous error
:cq	terminate with error condition
:cf	parse next set of errors

Colon Commands

:e <i>name</i>	edit file <i>name</i>
:e	redit last file
:e! <i>name</i>	edit file <i>name</i> , discarding changes
:e!	redit last file, discarding changes
:e #	edit alternate file
^^	edit alternate file
:e! #	edit alternate file, discarding changes
:fn	searches the file funclist or the environment variable FUNCLIST for a specified string; also invoked by typing ^_.
:r <i>name</i>	read file <i>name</i> into current file
:w	write back to file being edited
:wq	write back to file and quit
:w <i>name</i>	write to file <i>name</i> if does not exist
:w! <i>name</i>	write to file <i>name</i> , delete if exists

<code>:q</code>	quit
<code>:q!</code>	quit, discarding changes
<code>:x</code>	quit, saving file if modified
<code>ZZ</code>	quit, saving file if modified
<code>:f</code>	show current file and line
<code>:f <i>name</i></code>	change <i>name</i> of current file
<code>^G</code>	show current file and line
<code>:n</code>	edit next file in list
<code>:n!</code>	edit next file in list, discarding change
<code>:n <i>arg1 arg2....</i></code>	specify new list
<code>:rew</code>	point back to beginning of list
<code>:rew!</code>	point back to beginning, discarding changes
<code>:ta <i>tag</i></code>	position to <i>tag</i> in appropriate file, searches file pointed to by environment variable TAGS if tags does not exist or <i>tag</i> is not found in it.
<code>^]</code>	same as <code>:ta</code> using word at cursor
<code>:ta! <i>tag</i></code>	position to <i>tag</i> , discarding changes
<code>:>macro</code>	specify and execute immediate macro
<code>:set <i>opt1=val</i> <i>opt2=val ...</i></code>	set editor options
<code>:se <i>opt1=val</i> <i>opt2=val ...</i></code>	set editor options
<code>:set all</code>	display current option settings
<code>:[<i>range</i>]/<i>s/pat/rep</i>/[<i>options</i>]</code>	substitute <i>rep</i> format in <i>range</i>
<code>:[<i>range</i>]&</code>	repeat last substitute command

Index

\$,grep, 3-2
*,grep, 3-2 - 3-3
?,grep, 3-3

A

aborting make, 4-15
accessing files, Z editor, 5-39, 5-45
adding lines, diff, 2-2
AFLAGS macro, make, 4-11, 4-14
 value, 4-14
arcv, make, 4-20
arguments, grep, 3-2
assembly language source file, make, 4-21
autoindent, Z editor, 5-5, 5-28

B

backslash, make, 4-16
backspace key, Z editor, 5-5
backup files, Z editor, 5-6
batch commands, make, 4-14
blanks, diff, 2-4
buffer, Z editor, 5-7, 5-23 - 5-24
built-in rules, make, 4-14

C

CFLAGS macro, make, 4-11, 4-14
 value, 4-14
changing lines, diff, 2-2
character positioning commands, Z editor, 5-51
character strings
 make, 4-2, 4-9
 Z editor, 5-14
colon commands, Z editor, 5-53
command line, diff, 2-2

- command line, make, 4-15, 4-18
 - makefile, 4-10
 - maximum length, 4-16
 - optional parameters, 4-18
- command mode, Z editor, 5-6
- command sequence, make, 4-2
- components, Z editor, 5-3
- control keys, Z editor, 5-12
- conversion lists, diff, 2-1
 - types of operations, 2-2
- correction insert commands, Z editor, 5-52
- creating new program, Z editor, 5-3
- ctags utility, Z editor, 5-3, 5-45
- current directory, make, 4-17
- cursor motion commands, Z editor, 5-17
 - examples, 5-9
- cursor positioning, Z editor, 5-7

D

- default drive, make, 4-17
- definition
 - diff, 1-1
 - grep, 1-2
 - make, 1-2
 - makefile, 4-3
 - named buffers, Z editor, 5-24
 - regular expression, Z editor, 5-14
 - Z editor, 1-2, 5-1
- deleting lines, diff, 2-2
- deleting text, Z editor, 5-21 - 5-23
- dependency entries, make
 - definition, 4-3
 - makefile, 4-3, 4-10, 4-18
- dependency lines, make
 - maximum length, 4-17
- dependent files, make, 4-8 - 4-9
- Developer Level, 1-1
- diff
 - examples, 2-2 - 2-4
 - b option, 2-4

- adding lines, 2-2
- blanks, 2-4
- changing lines, 2-2
- command line, 2-2
- conversion items, 2-2 - 2-3
- conversion list, 2-1
- definition, 1-1
- deleting lines, 2-2
- directory names, 2-5
- file length, 2-4
- range of lines, 2-2
- replacing lines, 2-2
- UNIX options not supported, 2-4 - 2-5
- directory names, diff, 2-5
- directory, make
 - recording time and date, 4-3
- display commands, Z editor, 5-49
- duplicating text, Z editor, 5-23 - 5-24

E

- editing, Z editor, 5-5, 5-41 - 5-42
 - examples, 5-42
 - another file, 5-41
 - syntax, 5-41
- environment variable
 - ZOPT, 5-36
- escape key, Z editor, 5-12
- Ex-like commands, Z editor
 - "&" command, 5-35
 - addresses, 5-33
 - arguments, 5-32
 - c option, 5-34
 - g option, 5-34
 - repeat last substitute command, 5-32
 - substitute command, 5-32, 5-34

F

- file extension, make, 4-6
- file length, diff, 2-4

- file list feature, Z editor, 5-43
- file parameter, grep, 3-3
- file specification, grep, 3-2

G

- go command, Z editor, 5-8
- grep, 5-14
 - definition, 1-2, 3-2 - 3-3, 3-7 - 3-8
 - \$, 3-5
 - c option, 3-3
 - f option, 3-3
 - l option, 3-3
 - n option, 3-3
 - v option, 3-3
 - [[^]], 3-4
 - [], 3-5
 - \, 3-6
 - ^, 3-5
 - arguments, 3-2
 - file parameter, 3-3
 - file specification, 3-2
 - multiple file support, 3-2
 - multiple options, 3-8
 - patterns, 3-1 - 3-2
 - simple string matching, 3-7
 - special patterns examples, 3-5 - 3-6, 3-8
 - special patterns list, 3-4
 - standard input, 3-3
 - UNIX similarities and differences, 3-1, 3-7
 - wildcard characters, 3-2 - 3-3
 - x*, 3-6

I

- indirect macro definition, 5-31
- insert mode, Z editor, 5-10
 - ^W command, 5-5
 - command list, 5-26
 - commands, 5-27, 5-51
 - exiting, 5-5, 5-12

- i, 5-4
 - memory-resident buffer, 5-4
- interfile dependencies
 - makefile, 4-1

L

- line movement commands, Z editor, 5-15 - 5-17
- local moves, Z editor, 5-16
- logging commands, make, 4-15

M

- macro buffer, Z editor, 5-28
- macros, make, 4-9
 - examples, 4-10 - 4-12
 - \$\$, 4-11
 - \$\$*, 4-11
 - \$\$@, 4-11
 - AFLAGS, 4-11, 4-14
 - built-in rules, 4-11
 - capabilities, 4-2
 - CFLAGS, 4-11, 4-14
 - command line, 4-12
 - defining in command line, 4-11
 - invoking, 4-10
 - names, 4-2
 - naming, 4-10
 - option -DFLOAT, 4-12
 - used by built-in rules, 4-11
- macros, Z editor, 5-2, 5-29 - 5-32, 5-53
 - immediate macro definition, 5-28 - 5-30
 - indirect macro definition, 5-31
 - reexecuting, 5-31
- make
 - examples, 4-21
 - aborting, 4-15
 - advanced features, 4-8
 - arcv, 4-20
 - assembly language listing, 4-13
 - assembly language source file, 4-21

- backslash, 4-16
- batch commands, 4-14
- built-in rules, 4-11, 4-14
- C source file, 4-21
- character strings, 4-2, 4-9
- colon, 4-13
- command execution, 4-15
- command line, 4-12, 4-15, 4-18
- command line length, 4-15 - 4-16
- command line parameters, 4-18
- command sequence, 4-2
- comments, 4-16
- creating a makefile, 4-2
- current directory, 4-17
- default drive, 4-17
- default rules, 4-2
- definition, 1-2
- dependency lines, length, 4-17
- dependent files, 4-8 - 4-9
- disk file, 4-14
- file extension, 4-6
- filenames, 4-9
- interfile dependencies, 4-1
- line continuation, 4-16
- logging commands, 4-15
- macro definition example, 4-16
- makefile, 4-1
- mkarcv, 4-20
- naming conventions, 4-9
- null character string, 4-11, 4-14
- object files, 4-9
- object files, removing, 4-21
- operating system commands, 4-14
- overriding rules, 4-8
- parameters, 4-18
- path, 4-9
- prerequisite files, 4-3 - 4-4
- printing error messages, 4-18
- recording time and date, 4-3
- redirecting standard output, 4-18

- return code, 4-15
- rule definition, 4-12
- rule-processing capability, 4-2
- rules, 4-2, 4-6, 4-12
- rules built-in, 4-6
- rules, target extension, 4-6
- sequence of commands, rules, 4-6
- source extension, 4-12
- source extension, rules, 4-6
- special characters, examples, 4-15
- standard output, 4-18
- starting, 4-17, 4-21
- syntax, 4-17 - 4-18
- tab character, 4-13
- target extension, 4-12
- target file creation, 4-17
- target files, 4-3, 4-9
- UNIX options not supported, 4-19
- UNIX similarities and differences, 4-19
- volume, 4-9
- make built-in rules
 - examples, 4-7
 - .asm to .o rule, 4-7
 - .c to .o rule, 4-6 - 4-8
- make options
 - DDEBUG option, 4-14
 - t option, 4-14
 - b option, 4-19
 - d option, 4-19
 - e option, 4-19
 - i option, 4-19
 - k option, 4-19
 - m option, 4-19
 - n option, 4-19
 - q option, 4-19
 - r option, 4-19
 - s option, 4-19
 - t option, 4-19
- make, macros, 4-9
 - examples, 4-13

- AFLAGS, 4-11, 4-14
- CFLAGS, 4-11, 4-14
- command line, 4-12
- makefile
 - examples, 4-4 - 4-5, 4-20
 - f option, 4-3
 - command line, 4-10
 - definition, 4-3
 - dependency, 4-10
 - dependency entries, 4-3 - 4-5, 4-18
 - executed commands, 4-3
 - libc directory, 4-21
 - misc directory, 4-23
 - operating system commands, 4-3
 - syntax, 4-15
 - sys directory, 4-22
- marking and returning, Z editor, 5-19, 5-50
- matching patterns, Z editor, 5-14
- mkarcv, make, 4-20
- modifying text, Z editor, 5-21 - 5-22
- movement, Z editor
 - moving text, 5-23 - 5-25
 - within C programs, 5-18
 - within text, 5-8
 - word movement, 5-17 - 5-18
- multiple options, grep, 3-8

N

- named buffers, Z editor
 - advantage of, 5-24
 - definition, 5-24
 - moving text, 5-23
 - yanking text, 5-24
- notation conventions, 1-2
- null character string, make, 4-11, 4-14

O

- object files, make, 4-9
 - removing, 4-21

- operating system commands, make, 4-14
 - makefile, 4-3
- operator commands, Z editor, 5-52

P

- paging commands, Z editor, 5-12
- paragraph commands, Z editor, 5-51
- path, make, 4-9
- pattern searching, Z editor, 5-14 - 5-15
- patterns, grep, 3-1
 - special patterns, 3-4
 - within a file, 3-2
 - within multiple files, 3-2
- positioning, Z editor
 - line, 5-50
 - within files, 5-50
- prerequisite files, make, 4-3 - 4-4
- Professional Level, 1-1
- put commands, Z editor, 5-52

Q

- QuikFix, z editor, 5-1
 - e option, 5-6 - 5-7
 - o option, 5-7
 - :cc command, 5-36, 5-53
 - :cf command, 5-36, 5-53
 - :cn command, 5-36, 5-53
 - :cp command, 5-36, 5-53
 - :cq, 5-39
 - :cq command, 5-36, 5-53
 - :wq command, 5-39

R

- range of lines, diff, 2-2
- reading files command, Z editor, 5-41
- redo commands, Z editor, 5-53
- regular expression, Z editor, 5-14
 - definition, 5-14
 - special characters, 5-14

- special characters list, 5-15
- replace commands, Z editor, 5-51
- replacing lines, diff, 2-2
- requirements, Z editor, 5-2
- return code, make, 4-15
- rule-processing capability, make, 4-2
- rules, make, 4-2, 4-12
 - built-in, 4-6, 4-11, 4-14
 - definition, 4-12
 - makefile, 4-6
 - overriding, 4-8
 - sequence of commands, 4-6
 - source extension, 4-6
 - tab character, 4-13

S

- screen, Z editor, 5-11
 - adjusting, 5-49
 - display, 5-3
 - scrolling, 5-7
- shift operators, Z editor, 5-26
- shifting text, Z editor, 5-26
- simple string matching, grep, 3-7
- source extension, make, 4-12
- standard input, grep, 3-3
- standard output, make, 4-18
- starting, make, 4-21
- starting, Z editor, 5-38
- status information line, Z editor, 5-3
- stopping, Z editor, 5-39
- string search commands, Z editor, 5-8, 5-12 - 5-14
- substitute command, Z editor
 - examples, 5-35
 - options, 5-34
 - syntax, 5-34

T

- tab character, make, 4-13
- tags, Z editor, 5-44

- target extension, make, 4-12
 - rules, 4-6
- target file, Z editor, 5-25
- target files, make, 4-3, 4-9
 - creation, 4-17
- technical support, 1-3
- text, Z editor
 - line length, 5-11
 - rearranging, 5-24

U

- undo command, Z editor, 5-26, 5-53
- UNIX, 5-47
 - diff, 2-4
 - Ex-editor, 5-32
 - grep, 3-7
 - make, 4-1, 4-19
 - Vi editor, 5-1
- unnamed buffer, Z editor, 5-23
 - deleting text, 5-23
 - moving text, 5-23
- unprintable characters, Z editor, 5-11

V

- Vi editor, 5-47
- volume, make, 4-9

W

- wildcard characters, grep, 3-2 - 3-3
- word movement commands, Z editor, 5-18, 5-51
- wrap scan option, Z editor, 5-13
- writing files commands, Z editor, 5-40 - 5-41

Y

- yank operator command, Z editor, 5-23, 5-52

Z

Z editor

See also Z editor commands

.bak files, 5-6

accessing files, 5-39, 5-45

adjusting the screen, 5-20, 5-49 - 5-50

appending numbers to commands, 5-9

autoindent, 5-5, 5-28, 5-36

character positioning, 5-51

character strings, 5-14

colon commands, 5-13, 5-53

command mode, 5-6

components, 5-3

control keys, 5-12

corrections during insert, 5-52

creating new program, 5-3

ctags, 5-3, 5-45

cursor motion commands, 5-9, 5-18

cursor positioning, 5-7

definition, 1-2, 5-1

deleting lines, 5-10, 5-22

deleting text, 5-10, 5-21 - 5-22

differences between Z and Vi, 5-47 - 5-48

displaying unprintable characters, 5-11

duplicating text blocks, 5-23 - 5-24

echo commands, 5-12

editing another file, 5-41 - 5-43

editing existing files, 5-5

escape key, 5-5, 5-12

Ex-like commands, 5-32 - 5-35

exiting, 5-5

exiting insert mode, 5-5

file lists, 5-43

filenames, 5-40

indirect macro definition, 5-31

insert and replace, 5-51 - 5-52

insert command list, 5-26

insert commands, 5-10, 5-26 - 5-27

insert mode, 5-4, 5-27

insert mode, exiting, 5-12

- line movement, 5-15 - 5-17
- line positioning, 5-50
- macro buffer, 5-28
- macro wrap options, 5-32, 5-36
- macros, 5-2, 5-28 - 5-32, 5-53
- main text buffer, 5-23 - 5-24
- making changes, 5-20
- marking and returning, 5-19, 5-50
- matching patterns, 5-14 - 5-15
- miscellaneous operations, 5-52
- modifying text, 5-21 - 5-22
- movement within C programs, 5-18
- moving text between files, 5-24 - 5-25
- moving text blocks, 5-23
- moving within text, 5-8
- named buffers, 5-23 - 5-25
- operators, 5-52
- option file, 5-36 - 5-37
- options, 5-45 - 5-46, 5-49
- paging, 5-12
- positioning within files, 5-50
- QuikFix facility, 5-1
- reading files, 5-41
- rearranging text, 5-24
- reexecuting macros, 5-31
- regular expressions, 5-14
- requirements, 5-2
- screen display, 5-3 - 5-4
- scrolling, 5-7 - 5-8
- shift operators, 5-26
- shifting text, 5-26
- starting, 5-6, 5-37 - 5-38, 5-49
- status information line, 5-3
- stopping, 5-6, 5-39
- string search, 5-8, 5-12 - 5-13, 5-30
- substitute command, 5-34
- tags, 5-44
- text lines longer than screen, 5-11
- undo and redo, 5-53
- undo command, 5-26

- UNIX vi similarities and differences, 5-47 - 5-48
- unnamed buffer, 5-23 - 5-24
- unprintable characters, 5-11
- word movement, 5-17 - 5-18
- words and paragraphs, 5-51
- wrap scan option, 5-13
- writing files, 5-40 - 5-41
- yank and put, 5-52
- yank operator, 5-23 - 5-24
- yanking text, 5-24

Z editor command options

See also Z editor commands

- ai=1/0, 5-49
- ak, 5-49
- bk=1/0, 5-49
- cs, 5-46
- eb=1/0, 5-49
- ma=0/1, 5-49
- sm=1/0, 5-49
- ts=val, 5-49
- wm, 5-32
- wm=1/0, 5-49
- ws=1/0, 5-49

Z editor commands

See also Z editor, special character commands

- :se, 5-13
- a/A, 5-10, 5-26 - 5-27, 5-51
- b/B, 5-18, 5-30, 5-51
- backspace, 5-9, 5-27
- c/C, 5-21 - 5-22, 5-26, 5-52
- carriage return, 5-9, 5-50
- cc, 5-22
- Control L, 5-20
- Control-B, 5-12
- Control-D, 5-7, 5-27
- Control-F, 5-12
- Control-H, 5-16
- Control-J, 5-16
- Control-K, 5-16
- Control-L, 5-16

Control-U, 5-7
Control-V, 5-27
Control-X, 5-27
d/D, 5-21 - 5-22, 5-52
dd, 5-10, 5-20
dw, 5-22
 examples, deleting, 5-22
e/E, 5-18, 5-51
f/F, 5-17
fx/Fx, 5-51
g/G, 5-8, 5-50
H, 5-50
h/^H, 5-16, 5-51
i/I, 5-7, 5-26 - 5-27, 5-51
J, 5-52
j/^J, 5-16, 5-21, 5-50
k/^K, 5-16, 5-51
L, 5-50
l/^L, 5-16, 5-51
LF, 5-50
M, 5-50
mx, 5-19, 5-50
n/N, 5-9, 5-13, 5-50
o/O, 5-10, 5-17, 5-26, 5-52
p/P, 5-23, 5-25, 5-52
 examples, put, 5-25
R, 5-21, 5-52
rx, 5-21, 5-52
s/S, 5-21, 5-27, 5-52
space, 5-9, 5-51
t/T, 5-17
tabs, 5-49
tx/Tx, 5-51
u/U, 5-26, 5-53
v, 5-31, 5-53
w/W, 5-17 - 5-18, 5-30, 5-51
x/X, 5-7, 5-10, 5-20 - 5-21, 5-52
y, 5-23 - 5-24, 5-52
 examples, yank operator, 5-24
z/Z, 5-20, 5-49 - 5-50

- ZZ, 5-5 - 5-6, 5-39
- Z editor memory-resident buffer, 5-7
- Z editor, command line options
 - e option, 5-6 - 5-7
 - o option, 5-7
- Z editor, QuikFix facility
 - See QuikFix, Z editor
- Z editor, special character commands
 - See also Z editor commands
 - #, 5-40
 - \$, 5-15, 5-17
 - %, 5-18, 5-40
 - *, 5-15
 - /, 5-8
 - <, 5-15
 - <<, 5-52
 - >, 5-15
 - >>, 5-52
 - ?, 5-13
 - @, 5-11, 5-31
 - [[, 5-18
 - [], 5-14
 - [^str], 5-15
 - [str], 5-15
 - [x-y], 5-15
 - \, 5-40
 -]], 5-18
 - ^, 5-15, 5-17
 - {}, 5-19
 - |, 5-17
 - ~, 5-11
 - comma, 5-17
 - double quote, 5-19, 5-25
 - period, 5-15
 - semicolon, 5-17
 - single quote, 5-19
- ZOPT environment variable, Z editor, 5-36